Contents

1 Included Extensions 3

2 Website 5

3 Documentation 7

4 Source Code & Feedback 9

5 Questions and Answers 11

6 Contents 13
  6.1 Introduction ......................................................... 13
  6.2 Setup & Dependencies ............................................ 14
  6.3 Usage for Beginners .............................................. 16
  6.4 Basic Concepts .................................................... 21
  6.5 Tutorials .......................................................... 33
  6.6 Howto ............................................................. 129
  6.7 FAQ .............................................................. 136
  6.8 Reference .......................................................... 136
  6.9 Rendering ......................................................... 369
  6.10 Add-ons ........................................................... 387
  6.11 DXF Internals .................................................... 433
  6.12 Developer Guides ............................................... 513
  6.13 Glossary .......................................................... 530
  6.14 Indices and tables ............................................... 530

Python Module Index 531

Index 533
Welcome! This is the documentation for ezdxf release 0.16.2, last updated Apr 24, 2021.

- **ezdxf** is a Python package to create new DXF files and read/modify/write existing DXF files
- the intended audience are developers
- requires at least Python 3.7
- OS independent
- additional required packages: pyparsing
- optional Cython implementation of some low level math classes
- MIT-License
- additional read support for DXF versions R13/R14 (upgraded to R2000)
- additional read support for older DXF versions than R12 (upgraded to R12)
- read/write support for ASCII DXF and Binary DXF
- preserves third-party DXF content
Included Extensions

- **drawing** add-on to visualise and convert DXF files to images which can be saved to various formats such as png, pdf and svg
- **geo** add-on to support the `__geo_interface__`
- **r12writer** add-on to write basic DXF entities direct and fast into a DXF R12 file or stream
- **iterdxf** add-on to iterate over entities of the modelspace of really big (> 5GB) DXF files which do not fit into memory
- **importer** add-on to import entities, blocks and table entries from another DXF document
- **dxf2code** add-on to generate Python code for DXF structures loaded from DXF documents as starting point for parametric DXF entity creation
- **acadctb** add-on to read/write *Plot Style Files (CTB/STB)*
- **pycsg** add-on for Constructive Solid Geometry (CSG) modeling technique
Website

https://ezdxf.mozman.at/
CHAPTER 3

Documentation

Documentation of development version at https://ezdxf.mozman.at/docs
Documentation of latest release at http://ezdxf.readthedocs.io/
Source Code & Feedback

Source Code: http://github.com/mozman/ezdxl.git
Issue Tracker: http://github.com/mozman/ezdxl/issues
Forum: https://github.com/mozman/ezdxl/discussions
Questions and Answers

Please post questions at the forum or stack overflow to make answers available to other users as well.
6.1 Introduction

6.1.1 What is ezdxf

ezdxf is a Python interface to the DXF (drawing interchange file) format developed by Autodesk, ezdxf allows developers to read and modify existing DXF drawings or create new DXF drawings.

The main objective in the development of ezdxf was to hide complex DXF details from the programmer but still support most capabilities of the DXF format. Nevertheless, a basic understanding of the DXF format is required, also to understand which tasks and goals are possible to accomplish by using the the DXF format.

Not all DXF features are supported yet, but additional features will be added in the future gradually.

ezdxf is also a replacement for my dxfwrite and my dxfgrabber packages but with different APIs, for more information see also: What is the Relationship between ezdxf, dxfwrite and dxfgrabber?

6.1.2 What ezdxf can’t do

- ezdxf is not a DXF converter: ezdxf can not convert between different DXF versions, if you are looking for an appropriate application, try the free ODAFileConverter from the Open Design Alliance, which converts between different DXF version and also between the DXF and the DWG file format.

- ezdxf is not a CAD file format converter: ezdxf can not convert DXF files to other CAD formats such as DWG

- ezdxf is not a CAD kernel and does not provide high level functionality for construction work, it is just an interface to the DXF file format. If you are looking for a CAD kernel with Python scripting support, look at FreeCAD.

6.1.3 Supported Python Versions

ezdxf requires at least Python 3.7 and will be tested with the latest stable CPython version and the latest stable release of pypy3 during development.
ezdxf is written in pure Python with optional Cython implementations of some low level math classes and requires only pyparser as additional library beside the Python Standard Library. pytest is required to run the unit and integration tests. Data to run the stress and audit test can not be provided, because I don’t have the rights for publishing this DXF files.

### 6.1.4 Supported Operating Systems

ezdxf is OS independent and runs on all platforms which provide an appropriate Python interpreter (>=3.7).

### 6.1.5 Supported DXF Versions

<table>
<thead>
<tr>
<th>Version</th>
<th>AutoCAD Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC1009</td>
<td>AutoCAD R12</td>
</tr>
<tr>
<td>AC1012</td>
<td>AutoCAD R13 -&gt; R2000</td>
</tr>
<tr>
<td>AC1014</td>
<td>AutoCAD R14 -&gt; R2000</td>
</tr>
<tr>
<td>AC1015</td>
<td>AutoCAD R2000</td>
</tr>
<tr>
<td>AC1018</td>
<td>AutoCAD R2004</td>
</tr>
<tr>
<td>AC1021</td>
<td>AutoCAD R2007</td>
</tr>
<tr>
<td>AC1024</td>
<td>AutoCAD R2010</td>
</tr>
<tr>
<td>AC1027</td>
<td>AutoCAD R2013</td>
</tr>
<tr>
<td>AC1032</td>
<td>AutoCAD R2018</td>
</tr>
</tbody>
</table>

ezdxf also reads older DXF versions but saves it as DXF R12.

### 6.1.6 Embedded DXF Information of 3rd Party Applications

The DXF format allows third-party applications to embed application-specific information. ezdxf manages DXF data in a structure-preserving form, but for the price of large memory requirement. Because of this, processing of DXF information of third-party applications is possible and will retained on rewriting.

### 6.1.7 License

ezdxf is licensed under the very liberal MIT-License.

### 6.2 Setup & Dependencies

The primary goal is to keep the dependencies for the core package as small as possible. The add-ons are not part of the core package and can therefore use as many packages as needed. The only requirement for these packages is an easy way to install them on Windows, Linux and macOS, preferably as pip3 install ....

The pyparsing package is the only hard dependency and will be installed automatically by pip3!

Ezdxf provides since v0.15 some C-extensions, which will be deployed automatically at each release to PyPI as binary wheels for Windows, ManyLinux 2010 and macOS. The supported Python versions start with the latest stable version of pypy3, which is currently Python 3.7 (2021) and ends with the latest stable release of CPython.

The C-extensions are disabled for pypy3, because the JIT compiled code of pypy is much faster than the compiled C-extensions for pypy.
6.2.1 Basic Installation

The most common case is the installation by pip3 including the optional C-extensions from PyPI as binary wheels:

```
pip3 install ezdxf
```

6.2.2 Installation with Extras

To use all features of the drawing add-on, add the [draw] tag:

```
pip3 install ezdxf[draw]
```

<table>
<thead>
<tr>
<th>Tag</th>
<th>Additional Installed Packages</th>
</tr>
</thead>
<tbody>
<tr>
<td>[draw]</td>
<td>Matplotlib, PyQt5</td>
</tr>
<tr>
<td>[test]</td>
<td>geomdl, pytest</td>
</tr>
<tr>
<td>[dev]</td>
<td>setuptools, wheel, Cython + [test]</td>
</tr>
<tr>
<td>[all]</td>
<td>[draw] + [test] + [dev]</td>
</tr>
</tbody>
</table>

6.2.3 Disable C-Extensions

It is possible to disable the C-Extensions by setting the environment variable EZDXF_DISABLE_C_EXT to 1 or true:

```
set EZDXF_DISABLE_C_EXT=1
```

or on Linux:

```
export EZDXF_DISABLE_C_EXT=1
```

This is has to be done before anything from ezdxf is imported! If you are working in an interactive environment, you have to restart the interpreter.

6.2.4 Installation from GitHub

Install the latest development version by pip3 from GitHub:

```
pip3 install git+https://github.com/mozman/ezdxf.git@master
```

6.2.5 Build and Install from Source

Make a build directory and a virtual environment and sorry, I am working on Windows:

```
mkdir build
cd build
py -m venv py39
py39\Scripts\activate.bat
```

A working C++ compiler setup is required to compile the C-extensions from source code. Windows users need the build tools from Microsoft: https://visualstudio.microsoft.com/de/downloads/
Download and install the required Visual Studio Installer of the community edition and choose the option: Visual Studio Build Tools 20..

Install required packages to build and install ezdxf with C-extensions:

```bash
pip3 install setuptools wheel cython
```

Clone the GitHub repository:

```bash
git clone https://github.com/mozman/ezdxf.git
```

Build and install ezdxf from source code:

```bash
cd ezdxf
pip3 install .
```

Install the test dependencies and run the tests:

```bash
pip3 install pytest geomdl
pytest tests integration_tests
```

Install optional dependencies to use all features, like the drawing add-on:

```bash
pip3 install matplotlib PyQt5
```

### 6.2.6 Build Documentation

Assuming you are still in the build directory `build\ezdxf` of the previous section and matplotlib is installed.

Install Sphinx:

```bash
pip3 install Sphinx sphinx-rtd-theme
```

Build the HTML documentation:

```bash
cd docs
make.bat html
```

The output is located in `build\ezdxf\docs\build\html`.

### 6.3 Usage for Beginners

This section shows the intended usage of the `ezdxf` package. This is just a brief overview for new `ezdxf` users, follow the provided links for more detailed information.

First import the package:

```python
import ezdxf
```

#### 6.3.1 Loading DXF Files

`ezdxf` supports loading ASCII and binary DXF files from a file:
doc = ezdxf.readfile(filename)

or from a zip-file:

doc = ezdxf.readzip(zipfilename[, filename])

Which loads the DXF file filename from the zip-file zipfilename or the first DXF file in the zip-file if filename is absent.

It is also possible to read a DXF file from a stream by the ezdxf.read() function, but this is a more advanced feature, because this requires detection of the file encoding in advance.

This works well with DXF files from trusted sources like AutoCAD or BricsCAD, for loading DXF files with minor or major flaws look at the ezdxf.recover module.

See also:

documentation for ezdxf.readfile(), ezdxf.readzip() and ezdxf.read(), for more information about file management go to the Document Management section. For loading DXF files with structural errors look at the ezdxf.recover module.

6.3.2 Saving DXF Files

Save the DXF document with a new name:

doc.saveas('new_name.dxf')

or with the same name as loaded:

doc.save()

See also:


6.3.3 Create a New DXF File

Create new file for the latest supported DXF version:

doc = ezdxf.new()

Create a new DXF file for a specific DXF version, e.g for DXF R12:

doc = ezdxf.new('R12')

To setup some basic DXF resources use the setup argument:

doc = ezdxf.new(setup=True)

See also:

Documentation for ezdxf.new(), for more information about file management go to the Document Management section.
6.3.4 Layouts and Blocks

Layouts are containers for DXF entities like LINE or CIRCLE. The most important layout is the modelspace labeled as “Model” in CAD applications which represents the “world” workspace. Paperspace layouts represent plottable sheets which contain often the framing and the tile block of a drawing and VIEWPORT entities as scaled and clipped “windows” into the modelspace.

The modelspace is always present and can not be deleted. The active paperspace is also always present in a new DXF document but can be deleted, in that case another paperspace layout gets the new active paperspace, but you can not delete the last paperspace layout.

Getting the modelspace of a DXF document:
```
msp = doc.modelspace()
```

Getting a paperspace layout by the name as shown in the tab of a CAD application:
```
psp = doc.layout('Layout1')
```

A block is just another kind of entity space, which can be inserted multiple times into other layouts and blocks by the INSERT entity also called block references, this is a very powerful and important concept of the DXF format.

Getting a block layout by the block name:
```
blk = doc.blocks.get('NAME')
```

All these layouts have factory functions to create graphical DXF entities for their entity space, for more information about creating entities see section: Create new DXF Entities

6.3.5 Create New Blocks

The block definitions of a DXF document are managed by the `BlocksSection` object:
```
my_block = doc.blocks.new('MyBlock')
```

See also:

Tutorial for Blocks

6.3.6 Query DXF Entities

As said in the Layouts and Blocks section, all graphical DXF entities are stored in layouts, all these layouts can be iterated and support the index operator e.g. `layout[-1]` returns the last entity.

The main difference between iteration and index access is, that iteration filters destroyed entities, but the the index operator returns also destroyed entities until these entities are purged by `layout.purge()` more about this topic in section: Delete Entities.

There are two advanced query methods: `query()` and `groupby()`.

Get all lines of layer 'MyLayer':
```
lines = msp.query('LINE[layer=="MyLayer"]')
```

This returns an `EntityQuery` container, which also provides the same `query()` and `groupby()` methods.

Get all lines categorized by a DXF attribute like color:
all_lines_by_color = msp.query('LINE').groupby('color')
lines_with_color_1 = all_lines_by_color.get(1, [])

The `groupby()` method returns a regular Python `dict` with colors as key and a regular Python `list` of entities as values (not an `EntityQuery` container).

See also:

For more information go to the *Tutorial for getting data from DXF files*

### 6.3.7 Examine DXF Entities

Each DXF entity has a `dxf` namespace attribute, which stores the named DXF attributes, some DXF attributes are only indirect available like the vertices in the LWPOLYLINE entity. More information about the DXF attributes of each entity can found in the documentation of the `ezdxf.entities` module.

Get some basic DXF attributes:

```python
layer = entity.dxf.layer  # default is '0'
color = entity.dxf.color  # default is 256 = BYLAYER
```

Most DXF attributes have a default value, which will be returned if the DXF attribute is not present, for DXF attributes without a default value you can check in the attribute really exist:

```python
eentity.dxf.hasattr('true_color')
```

or use the `get()` method and a default value:

```python
eentity.dxf.get('true_color', 0)
```

See also:

*Common graphical DXF attributes*

### 6.3.8 Create New DXF Entities

The factory functions for creating new graphical DXF entities are located in the `BaseLayout` class. This means this factory function are available for all entity containers:

- `Modelspace`
- `Paperspace`
- `BlockLayout`

The usage is simple:

```python
msp = doc.modelspace()
msp.add_line((0, 0), (1, 0), dxfattribs={'layer': 'MyLayer'})
```

A few important or required DXF attributes are explicit method arguments, most additional and optional DXF attributes are gives as a regular Python `dict` object. The supported DXF attributes can be found in the documentation of the `ezdxf.entities` module.

*Warning:* Do not instantiate DXF entities by yourself and add them to layouts, always use the provided factory function to create new graphical entities, this is the intended way to use *ezdxf*. 

### 6.3. Usage for Beginners 19
6.3.9 Create Block References

A block reference is just another DXF entity called INSERT, but the term “Block Reference” is a better choice and so the Insert entity is created by the factory function: add_blockref():

msp.add_blockref('MyBlock')

See also:
See Tutorial for Blocks for more advanced features like using Attrib entities.

6.3.10 Create New Layers

A layer is not an entity container, a layer is just another DXF attribute stored in the entity and this entity can inherit some properties from this Layer object. Layer objects are stored in the layer table which is available as attribute doc.layers.

You can create your own layers:

my_layer = doc.layer.new('MyLayer')

The layer object also controls the visibility of entities which references this layer, the on/off state of the layer is unfortunately stored as positive or negative color value which make the raw DXF attribute of layers useless, to change the color of a layer use the property Layer.color

my_layer.color = 1

To change the state of a layer use the provided methods of the Layer object, like on(), off(), freeze() or thaw():

my_layer.off()

See also:
Layer Concept

6.3.11 Delete Entities

The safest way to delete entities is to delete the entity from the layout containing that entity:

line = msp.add_line((0, 0), (1, 0))
msp.delete_entity(line)

This removes the entity immediately from the layout and destroys the entity. The property is_alive returns False for a destroyed entity and all Python attributes are deleted, so line.dxf.color will raise an AttributeError exception, because line does not have a dxf attribute anymore.

The current version of ezdxf also supports also destruction of entities by calling method destroy() manually:

line.destroy()

Manually destroyed entities are not removed immediately from entities containers like Modelspace or EntityQuery, but iterating such a container will filter destroyed entities automatically, so a for e in msp: ... loop will never yield destroyed entities. The index operator and the len() function do not filter deleted entities, to avoid getting deleted entities call the purge() method of the container manually to remove deleted entities.
6.3.12 Further Information

• Reference documentation
• Documentation of package internals: Developer Guides.

6.4 Basic Concepts

The Basic Concepts section teach the intended meaning of DXF attributes and structures without teaching the application of this information or the specific implementation by ezdxf, if you are looking for more information about the ezdxf internals look at the Reference section or if you want to learn how to use ezdxf go to the Tutorials section and for the solution of specific problems go to the Howto section.

6.4.1 What is DXF?

The common assumption is also the cite of Wikipedia:

AutoCAD DXF (Drawing eXchange Format) is a CAD data file format developed by Autodesk for enabling data interoperability between AutoCAD and other applications.

DXF was originally introduced in December 1982 as part of AutoCAD 1.0, and was intended to provide an exact representation of the data in the AutoCAD native file format, DWG (Drawing). For many years Autodesk did not publish specifications making correct imports of DXF files difficult. Autodesk now publishes the DXF specifications online.

The more precise cite from the DXF reference itself:

The DXF™ format is a tagged data representation of all the information contained in an AutoCAD® drawing file. Tagged data means that each data element in the file is preceded by an integer number that is called a group code. A group code’s value indicates what type of data element follows. This value also indicates the meaning of a data element for a given object (or record) type. Virtually all user-specified information in a drawing file can be represented in DXF format.

No mention of interoperability between AutoCAD and other applications.

In reality the DXF format was designed to ensure AutoCAD cross-platform compatibility in the early days when different hardware platforms with different binary data formats were used. The name DXF (Drawing eXchange Format) may suggest an universal exchange format, but it is not. It is based on the infrastructure installed by Autodesk products (fonts) and the implementation details of AutoCAD (MTEXT) or on licensed third party technologies (embedded ACIS entities).

For more information about the AutoCAD history see the document: The Autodesk File - Bits of History, Words of Experience by John Walker, founder of Autodesk, Inc. and co-author of AutoCAD.

DXF Reference Quality

The DXF reference is by far no specification nor a standard like the W3C standard for SVG or the ISO standard for PDF.

The reference describes many but not all DXF entities and some basic concepts like the tag structure or the arbitrary axis algorithm. But the existing documentation (reference) is incomplete and partly misleading or wrong. Also missing from the reference are some important parts like the complex relationship between the entities to create higher order structures like block definitions, layouts (model space & paper space) or dynamic blocks to name a few.
### 6.4.2 AutoCAD Color Index (ACI)

The `color` attribute represents an ACI (AutoCAD Color Index). AutoCAD and many other CAD application provides a default color table, but pen table would be the more correct term. Each ACI entry defines the color value, the line weight and some other attributes to use for the pen. This pen table can be edited by the user or loaded from an `CTB` or `STB` file. *ezdxf* provides functions to create `(new())` or modify `(ezdxf.acadctb.load())` plot styles files.

DXF R12 and prior are not good in preserving the layout of a drawing, because of the lack of a standard color table defined by the DXF reference and missing DXF structures to define these color tables in the DXF file. So if a CAD user redefined an ACI and do not provide a `CTB` or `STB` file, you have no ability to determine which color or lineweight was used. This is better in later DXF versions by providing additional DXF attributes like `lineweight` and `true_color`.

See also:

*Plot Style Files (CTB/STB)*

### 6.4.3 Layer Concept

Every object has a layer as one of its properties. You may be familiar with layers - independent drawing spaces that stack on top of each other to create an overall image - from using drawing programs. Most CAD programs use layers as the primary organizing principle for all the objects that you draw. You use layers to organize objects into logical groups of things that belong together; for example, walls, furniture, and text notes usually belong on three separate layers, for a couple of reasons:

- Layers give you a way to turn groups of objects on and off - both on the screen and on the plot.
- Layers provide the most efficient way of controlling object color and linetype.

Create a layer table entry `Layer` by `Drawing.layers.new()`, assign the layer properties such as color and linetype. Then assign those layers to other DXF entities by setting the DXF attribute `layer` to the layer name as string.

It is possible to use layers without a layer definition but not recommend, just use a layer name without a layer definition, the layer has the default linetype 'Continuous' and the default color is 7.

The advantage of assigning a linetype and a color to a layer is that entities on this layer can inherit this properties by using 'BYLAYER' as linetype string and 256 as color, both values are default values for new entities.

See also:

*Tutorial for Layers*

### 6.4.4 Linetypes

The `linetype` defines the pattern of a line. The linetype of an entity can be specified by the DXF attribute `linetype`, this can be an explicit named linetype or the entity can inherit its line type from the assigned layer by setting `linetype` to 'BYLAYER', which is also the default value. CONTINUOUS is the default line type for layers with unspecified line type.

*ezdxf* creates several standard linetypes, if the argument `setup` is `True` at calling `new()`, this simple line types are supported by all DXF versions:

```python
doc = ezdxf.new('R2007', setup=True)
```
In DXF R13 Autodesk introduced complex linetypes, containing TEXT or SHAPES in linetypes. `ezdxf` v0.8.4 and later supports complex linetypes.

See also:

* Tutorial for Linetypes

**Linetype Scaling**

Global linetype scaling can be changed by setting the header variable `doc.header['$LTSCALE'] = 2`, which stretches the line pattern by factor 2.

To change the linetype scaling for single entities set scaling factor by DXF attribute `ltscale`, which is supported since DXF version R2000.

### 6.4.5 Coordinate Systems

AutoLISP Reference to Coordinate Systems provided by Autodesk.

To brush up your knowledge about vectors, watch the YouTube tutorials of 3Blue1Brown about Linear Algebra.

**WCS**

World coordinate system - the reference coordinate system. All other coordinate systems are defined relative to the WCS, which never changes. Values measured relative to the WCS are stable across changes to other coordinate systems.

**UCS**

User coordinate system - the working coordinate system defined by the user to make drawing tasks easier. All points passed to AutoCAD commands, including those returned from AutoLISP routines and external functions, are points in the current UCS. As far as I know, all coordinates stored in DXF files are always WCS or OCS never UCS.

User defined coordinate systems are not just helpful for interactive CAD, therefore `ezdxf` provides a converter class `UCS` to translate coordinates from UCS into WCS and vice versa, but always remember: store only WCS or OCS coordinates in DXF files, because there is no method to determine which UCS was active or used to create UCS coordinates.

See also:

- Table entry `UCS`
- `ezdxf.math.UCS` - converter between WCS and UCS

**OCS**

Object coordinate system - coordinates relative to the object itself. These points are usually converted into the WCS, current UCS, or current DCS, according to the intended use of the object. Conversely, points must be translated into an OCS before they are written to the database. This is also known as the entity coordinate system.

Because `ezdxf` is just an interface to DXF, it does not automatically convert OCS into WCS, this is the domain of the user/application. And further more, the main goal of OCS is to place 2D elements in 3D space, this maybe was useful in the early years of CAD, I think nowadays this is an not often used feature, but I am not an AutoCAD user.

OCS differ from WCS only if extrusion != (0, 0, 1), convert OCS into WCS:
# circle is an DXF entity with extrusion != (0, 0, 1)
ocs = circle.ocs()
wcs_center = ocs.to_wcs(circle.dxf.center)

See also:
- **Object Coordinate System (OCS)** - deeper insights into OCS
- **ezdxf.math.OCS** - converter between WCS and OCS

## DCS

Display coordinate system - the coordinate system into which objects are transformed before they are displayed. The origin of the DCS is the point stored in the AutoCAD system variable TARGET, and its z-axis is the viewing direction. In other words, a viewport is always a plan view of its DCS. These coordinates can be used to determine where something will be displayed to the AutoCAD user.

### 6.4.6 Object Coordinate System (OCS)

- **DXF Reference for OCS** provided by Autodesk.

The points associated with each entity are expressed in terms of the entity’s own object coordinate system (OCS). The OCS was referred to as ECS in previous releases of AutoCAD.

With OCS, the only additional information needed to describe the entity’s position in 3D space is the 3D vector describing the z-axis of the OCS, and the elevation value.

For a given z-axis (or extrusion) direction, there are an infinite number of coordinate systems, defined by translating the origin in 3D space and by rotating the x- and y-axis around the z-axis. However, for the same z-axis direction, there is only one OCS. It has the following properties:

- Its origin coincides with the WCS origin.
- The orientation of the x- and y-axis within the xy-plane are calculated in an arbitrary but consistent manner. AutoCAD performs this calculation using the arbitrary axis algorithm.

These entities do not lie in a particular plane. All points are expressed in world coordinates. Of these entities, only lines and points can be extruded. Their extrusion direction can differ from the world z-axis.

- **Line**
- **Point**
- **3DFace**
- **Polyline** (3D)
- **Vertex** (3D)
- **Polymesh**
- **Polyface**
- **Viewport**

These entities are planar in nature. All points are expressed in object coordinates. All of these entities can be extruded. Their extrusion direction can differ from the world z-axis.

- **Circle**
- **Arc**
Some of a Dimension’s points are expressed in WCS and some in OCS.

**Elevation**

Elevation group code 38:

Exists only in output from versions prior to R11. Otherwise, Z coordinates are supplied as part of each of the entity’s defining points.

**Arbitrary Axis Algorithm**

The arbitrary axis algorithm is used by AutoCAD internally to implement the arbitrary but consistent generation of object coordinate systems for all entities that use object coordinates.

Given a unit-length vector to be used as the z-axis of a coordinate system, the arbitrary axis algorithm generates a corresponding x-axis for the coordinate system. The y-axis follows by application of the right-hand rule.

We are looking for the arbitrary x- and y-axis to go with the normal Az (the arbitrary z-axis). They will be called Ax and Ay (using Vec3):

```python
Az = Vec3(entity.dxf.extrusion).normalize()  # normal (extrusion) vector
    # Extrusion vector normalization should not be necessary, but don't rely on any DXF content
if (abs(Az.x) < 1/64.) and (abs(Az.y) < 1/64.):
    Ax = Vec3(0, 1, 0).cross(Az).normalize()  # the cross-product operator
else:
    Ax = Vec3(0, 0, 1).cross(Az).normalize()  # the cross-product operator
Ay = Az.cross(Ax).normalize()
```

**WCS to OCS**

26 Chapter 6. Contents
**OCS to WCS**

```python
def wcs_to_ocs(point):
    px, py, pz = Vec3(point)  # point in WCS
    x = px * Ax.x + py * Ax.y + pz * Ax.z
    y = px * Ay.x + py * Ay.y + pz * Ay.z
    z = px * Az.x + py * Az.y + pz * Az.z
    return Vec3(x, y, z)
```

```python
Wx = wcs_to_ocs((1, 0, 0))
Wy = wcs_to_ocs((0, 1, 0))
Wz = wcs_to_ocs((0, 0, 1))
```

```python
def ocs_to_wcs(point):
    px, py, pz = Vec3(point)  # point in OCS
    x = px * Wx.x + py * Wx.y + pz * Wx.z
    y = px * Wy.x + py * Wy.y + pz * Wy.z
    z = px * Wz.x + py * Wz.y + pz * Wz.z
    return Vec3(x, y, z)
```

**6.4.7 DXF Units**

The DXF reference has no explicit information how to handle units in DXF, any information in this section is based on experiments with BricsCAD and may differ in other CAD application, BricsCAD tries to be as compatible with AutoCAD as possible. Therefore, this information should also apply to AutoCAD.

Please open an issue on github if you have any corrections or additional information about this topic.

**Length Units**

Any length or coordinate value in DXF is unitless in the first place, there is no unit information attached to the value. The unit information comes from the context where a DXF entity is used. The document/modelspace get the unit information from the header variable $INSUNITS, paperspace and block layouts get their unit information from the attribute units. The modelspace object has also a units property, but this value do not represent the modelspace units, this value is always set to 0 “unitless”.

Get and set document/modelspace units as enum by the Drawing property units:

```python
import ezdxf
from ezdxf import units
doc = ezdxf.new()
# Set centimeter as document/modelspace units
doc.units = units.CM
# which is a shortcut (including validation) for
doc.header['$INSUNITS'] = units.CM
```

**Block Units**

As said each block definition can have independent units, but there is no implicit unit conversion applied, not in CAD applications and not in ezdxf.
When inserting a block reference (INSERT) into the modelspace or another block layout with different units, the scaling factor between these units must be applied explicit as scaling DXF attributes (xscale, ...) of the \textit{Insert} entity, e.g. modelspace in meters and block in centimeters, x-, y- and z-scaling has to be 0.01:

```python
doc.units = units.M
my_block = doc.blocks.new('MYBLOCK')
my_block.units = units.CM
block_ref = msp.add_block_ref('MYBLOCK')
# Set uniform scaling for x-, y- and z-axis
block_ref.set_scale(0.01)
```

Use helper function \texttt{conversion\_factor()} to calculate the scaling factor between units:

```python
factor = units.conversion_factor(doc.units, my_block.units)
# factor = 100 for 1m is 100cm
# scaling factor = 1 / factor
block_ref.set_scale(1.0/factor)
```

**Hint:** It is never a good idea to use different measurement system in one document, ask the NASA about their Mars Climate Orbiter from 1999. The same applies for units of the same measurement system, just use one unit like meters or inches.

### Angle Units

Angles are always in degrees (360 deg = full circle) and in counter clockwise orientation, unless stated explicit otherwise.

### Display Format

How values are shown in the CAD GUI is controlled by the header variables \$\text{LUNITS} and \$\text{AUNITS}, but this has no meaning for values stored in DXF files.

### \$\text{INSUNITS}

The most important setting is the header variable \$\text{INSUNITS}, this variable defines the drawing units for the modelspace and therefore for the DXF document if no further settings are applied.

The modelspace \texttt{LAYOUT} entity has a property \texttt{units} as any layout like object, but it seem to have no meaning for the modelspace, BricsCAD set this property always to 0, which means unitless.

The most common units are 6 for meters and 1 for inches.

```python
doc.header['$\text{INSUNITS}'] = 6
```
The header variable `$MEASUREMENT` controls whether the current drawing uses imperial or metric hatch pattern and linetype files, this setting is not applied correct in ezdxf yet, but will be fixed in the future:

This setting is independent from `$INSUNITS`, it is possible to set the drawing units to inch and use metric linetypes and hatch pattern.

In BricsCAD the base scaling of the linetypes is only depending from the `$MEASUREMENT` value, is not relevant if `$INSUNITS` is meter, centimeter, millimeter, … and so on and the same is valid for hatch pattern.

```python
doc.header['$MEASUREMENT'] = 1
```

### $LUNITS

The header variable `$LUNITS` defines how CAD applications show linear values in the GUI and has no meaning for ezdxf:

```python
doc.header['$LUNITS'] = 2
```
$AUNITS$

The header variable $AUNITS$ defines how CAD applications show angular values in the GUI and has no meaning for ezdxf, DXF angles are always degrees in counter-clockwise orientation, unless stated explicit otherwise:

```python
doc.header['$AUNITS$'] = 0
```

### Helper Tools

ezdxf.units.\texttt{conversion_factor}(\texttt{source\_units: int, target\_units: int}) \rightarrow float

Returns the conversion factor to represent \texttt{source\_units} in \texttt{target\_units}.

E.g. millimeter in centimeter `conversion_factor(MM, CM)` returns 0.1, because 1 mm = 0.1 cm

### 6.4.8 Layout Extents and Limits

The \textit{extents} and \textit{limits} of an layout represents borders which can be referenced by the ZOOM command or read from some header variables from the \textit{HeaderSection}, if the creator application maintains these values – ezdxf does it not automatically.

#### Extents

The \textit{extents} of an layout are determined by the maximum extents of all DXF entities that are in this layout. The command:

```python
ZOOM extents
```

sets the current viewport to the extents of the currently selected layout.

A paper space layout in an arbitrary zoom state:
Limits

Sets an invisible rectangular boundary in the drawing area that can limit the grid display and limit clicking or entering point locations. The default limits for paper space layouts is defined by the paper size.
The layout from above after the ZOOM all command:

See also:
The AutoCAD online reference for the ZOOM and the LIMITS command.

**Read Stored Values**

The extents of the model space (the tab called “Model”) are stored in the header variable $\texttt{EXTMIN}$ and $\texttt{EXTMAX}$. The default values of $\texttt{EXTMIN}$ is $(+1e20, +1e20, +1e20)$ and $\texttt{EXTMAX}$ is $(-1e20, -1e20, -1e20)$, which do not describe real borders. These values are copies of the extents attributes of the Layout object as Layout.dxf.extmin and Layout.dxf.extmax.

The limits of the modelspace are stored in the header variables $\texttt{LIMMIN}$ and $\texttt{LIMMAX}$ and have default values of $(0, 0)$ and $(420, 297)$, the default paper size of ezdxf in drawing units. These are copies of the Layout attributes Layout.dxf.extmin and Layout.dxf.extmax.

The extents and the limits of the actual paper space layout, which is the last activated paper space layout tab, stored in the header variable $\texttt{PEXTMIN}$, $\texttt{PEXTMAX}$, $\texttt{PLIMMIN}$ and $\texttt{PLIMMAX}$.

Each paper space layout has its own values stored in the Layout attributes Layout.dxf.extmin, Layout.dxf.extmax, Layout.dxf.limmin and Layout.dxf.limax.

**Setting Extents and Limits**

Since v0.16 ezdxf it is sufficient to define the attributes for extents and limits (Layout.dxf.extmax, Layout.dxf.limmin and Layout.dxf.limax) of Layout object. The header variables are synchronized when the document is saved.

The extents of a layout are not calculated automatically by ezdxf, as this can take a long time for large documents and correct values are not required to create a valid DXF document.

See also:
How to: Calculate Extents for the Modelspace

6.4.9 Font Resources

DXF relies on the infrastructure installed by AutoCAD like the included SHX files or True Type fonts. There is no simple way to store additional information about a used fonts beside the plain file system name like "arial.ttf". The CAD application or viewer which opens the DXF file has to have access to the specified fonts used in your DXF document or it has to use an appropriate replacement font, which is not that easy in the age of unicode. Later DXF versions can store font family names in the XDATA of the STYLE entity but not all CAD application use this information.

6.5 Tutorials

6.5.1 Tutorial for getting data from DXF files

In this tutorial I show you how to get data from an existing DXF drawing.

Loading the DXF file:

```python
import sys
import ezdxf

try:
    doc = ezdxf.readfile("your_dxf_file.dxf")
except IOError:
    print(f'Not a DXF file or a generic I/O error.
sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file.
    sys.exit(2)
```

This works well with DXF files from trusted sources like AutoCAD or BricsCAD, for loading DXF files with minor or major flaws look at the `ezdxf.recover` module.

See also:

* Document Management

Layouts

I use the term layout as synonym for an arbitrary entity space which can contain DXF entities like LINE, CIRCLE, TEXT and so on. Every DXF entity can only reside in exact one layout.

There are three different layout types:

- **Modelspace**: this is the common construction space
- **Paperspace**: used to to create print layouts
- **BlockLayout**: reusable elements, every block has its own entity space

A DXF drawing consist of exact one modelspace and at least of one paperspace. DXF R12 has only one unnamed paperspace the later DXF versions support more than one paperspace and each paperspace has a name.
Iterate over DXF entities of a layout

Iterate over all DXF entities in modelspace. Although this is a possible way to retrieve DXF entities, I would like to point out that entity queries are the better way.

```python
# helper function
def print_entity(e):
    print("LINE on layer: %s\n" % e.dxf.layer)
    print("start point: %s\n" % e.dxf.start)
    print("end point: %s\n" % e.dxf.end)

# iterate over all entities in modelspace
msp = doc.modelspace()
for e in msp:
    if e.dxftype() == 'LINE':
        print_entity(e)

# entity query for all LINE entities in modelspace
for e in msp.query('LINE'):
    print_entity(e)
```

All layout objects supports the standard Python iterator protocol and the in operator.

Access DXF attributes of an entity

Check the type of an DXF entity by `e.dxftype()`. The DXF type is always uppercase. All DXF attributes of an entity are grouped in the namespace attribute `dxf`:

```python
e.dxf.layer  # layer of the entity as string
e.dxf.color  # color of the entity as integer
```

See Common graphical DXF attributes

If a DXF attribute is not set (a valid DXF attribute has no value), a DXFValueError will be raised. To avoid this use the `get_dxf_attrib()` method with a default value:

```python
# If DXF attribute 'paperspace' does not exist, the entity defaults
# to modelspace:
p = e.get_dxf_attrib('paperspace', 0)
```

An unsupported DXF attribute raises an DXFAttributeError.

Getting a paperspace layout

```python
paperspace = doc.layout('layout0')
```

Retrieves the paperspace named `layout0`, the usage of the `Layout` object is the same as of the modelspace object. DXF R12 provides only one paperspace, therefore the paperspace name in the method call `doc.layout('layout0')` is ignored or can be left off. For the later DXF versions you get a list of the names of the available layouts by `layout_names()`.

Retrieve entities by query language

`ezdxf` provides a flexible query language for DXF entities. All layout types have a `query()` method to start an entity query or use the `ezdxf.query.new()` function.
The query string is the combination of two queries, first the required entity query and second the optional attribute query, enclosed in square brackets: 'EntityQuery[AttributeQuery]'

The entity query is a whitespace separated list of DXF entity names or the special name *. Where * means all DXF entities, all other DXF names have to be uppercase. The * search can exclude entity types by adding the entity name with a presceding ! (e.g. * !LINE, search all entities except lines).

The attribute query is used to select DXF entities by its DXF attributes. The attribute query is an addition to the entity query and matches only if the entity already match the entity query. The attribute query is a boolean expression, supported operators: and, or, !.

See also:

* Entity Query String

Get all LINE entities from the modelspace:

```python
msp = doc.modelspace()
lines = msp.query('LINE')
```

The result container `EntityQuery` also provides the `query()` method, get all LINE entities at layer construction:

```python
construction_lines = lines.query('*[layer=="construction"]')
```

The * is a wildcard for all DXF types, in this case you could also use LINE instead of *, * works here because lines just contains entities of DXF type LINE.

All together as one query:

```python
lines = msp.query('LINE[layer=="construction"]')
```

The ENTITIES section also supports the `query()` method:

```python
lines_and_circles = doc.entities.query('LINE CIRCLE[layer=="construction"]')
```

Get all modelspace entities at layer construction, but excluding entities with linetype DASHED:

```python
not_dashed_entities = msp.query('*[layer=="construction" and linetype!="DASHED"]')
```

**Retrieve entities by groupby() function**

Search and group entities by a user defined criteria. As example let’s group all entities from modelspace by layer, the result will be a dict with layer names as dict-key and a list of all entities from modelspace matching this layer as dict-value. Usage as dedicated function call:

```python
from ezdxf.groupby import groupby
group = groupby(entities=msp, dxfattrib='layer')
```

The `entities` argument can be any container or generator which yields `DXFEntity` or inherited objects. Shorter and simpler to use as method of `BaseLayout` (modelspace, paperspace layouts, blocks) and query results as `EntityQuery` objects:

```python
group = msp.groupby(dxfattrib='layer')
for layer, entities in group.items():
    print(f'Layer "/{layer}" contains following entities:"
```
(continues on next page)
The previous example shows how to group entities by a single DXF attribute, but it is also possible to group entities by a custom key, to do so create a custom key function, which accepts a DXF entity as argument and returns a hashable value as dict-key or None to exclude the entity. The following example shows how to group entities by layer and color, so each result entry has a tuple (layer, color) as key and a list of entities with matching DXF attributes:

```python
def layer_and_color_key(entity):
    # return None to exclude entities from result container
    if entity.dxf.layer == '0':  # exclude entities from default layer '0'
        return None
    else:
        return entity.dxf.layer, entity.dxf.color

group = msp.groupby(key=layer_and_color_key)
for key, entities in group.items():
    print(f'Grouping criteria "{key}" matches following entities:)
    for entity in entities:
        print(' ' + str(entity))
    print('-' * 40)
```

To exclude entities from the result container the key function should return None. The groupby() function catches DXFAtributeError exceptions while processing entities and excludes this entities from the result container. So there is no need to worry about DXF entities which do not support certain attributes, they will be excluded automatically.

See also:

- groupby() documentation

### 6.5.2 Tutorial for creating simple DXF drawings

**r12writer** - create simple DXF R12 drawings with a restricted entities set: LINE, CIRCLE, ARC, TEXT, POINT, SOLID, 3DFACE and POLYLINE. Advantage of the r12writer is the speed and the low memory footprint, all entities are written direct to the file/stream without building a drawing data structure in memory.

See also:

- r12writer

Create a new DXF drawing with ezdxf.new() to use all available DXF entities:

```python
import ezdxf

doc = ezdxf.new('R2010')  # create a new DXF R2010 drawing, official DXF version
msp = doc.modelspace()  # add new entities to the modelspace
msp.add_line((0, 0), (10, 0))  # add a LINE entity
doc.saveas('line.dxf')
```

New entities are always added to layouts, a layout can be the modelspace, a paperspace layout or a block layout.

See also:

Look at factory methods of the BaseLayout class to see all the available DXF entities.
6.5.3 Tutorial for Layers

If you are not familiar with the concept of layers, please read this first: *Layer Concept*

Create a Layer Definition

```python
import ezdxf

doc = ezdxf.new(setup=True)  # setup required line types
msp = doc.modelspace()
doc.layers.new(name='MyLines', dxfattribs={'linetype': 'DASHED', 'color': 7})
```

The advantage of assigning a linetype and a color to a layer is that entities on this layer can inherit this properties by using 'BYLAYER' as linetype string and 256 as color, both values are default values for new entities so you can leave off these assignments:

```python
msp.add_line((0, 0), (10, 0), dxfattribs={'layer': 'MyLines'})
```

The new created line will be drawn with color 7 and linetype 'DASHED'.

Changing Layer State

Get the layer definition object:

```python
my_lines = doc.layers.get('MyLines')
```

Check the state of the layer:

```python
my_lines.is_off()  # True if layer is off
my_lines.is_on()   # True if layer is on
my_lines.is_locked()  # True if layer is locked
layer_name = my_lines.dxf.name  # get the layer name
```

Change the state of the layer:

```python
# switch layer off, entities at this layer will not shown in CAD applications/viewers
my_lines.off()

# lock layer, entities at this layer are not editable in CAD applications
my_lines.lock()
```

Get/set default color of a layer by property `Layer.color`, because the DXF attribute `Layer.dxf.color` is misused for switching the layer on and off, layer is off if the color value is negative.

Changing the default layer values:

```python
my_lines.dxf.linetype = 'DOTTED'
my_lines.color = 13  # preserves on/off state of layer
```

See also:

For all methods and attributes see class *Layer.*
Check Available Layers

The layers object supports some standard Python protocols:

```python
# iteration
for layer in doc.layers:
    if layer.dxf.name != '0':
        layer.off()  # switch all layers off except layer '0'

# check for existing layer definition
if 'MyLines' in doc.layers:
    layer = doc.layers.get('MyLines')

layer_count = len(doc.layers)  # total count of layer definitions
```

Deleting a Layer

Delete a layer definition:

```python
doc.layers.remove('MyLines')
```

This just deletes the layer definition, all DXF entities with the DXF attribute layer set to 'MyLines' are still there, but if they inherit color and/or linetype from the layer definition they will be drawn now with linetype 'Continuous' and color 1.

6.5.4 Tutorial for Blocks

What are Blocks?

Blocks are collections of DXF entities which can be placed multiple times as block references in different layouts and other block definitions. The block reference (Insert) can be rotated, scaled, placed in 3D by OCS and arranged in a grid like manner, each Insert entity can have individual attributes (Attrib) attached.

Create a Block

Blocks are managed as BlockLayout by a BlocksSection object, every drawing has only one blocks section stored in the attribute: Drawing.blocks.

```python
import ezdxf
import random  # needed for random placing points

def get_random_point():
    """Returns random x, y coordinates.""
    x = random.randint(-100, 100)
    y = random.randint(-100, 100)
    return x, y

# Create a new drawing in the DXF format of AutoCAD 2010
doc = ezdxf.new('R2010')

# Create a block with the name 'FLAG'
```
Block References (Insert)

A block reference is a DXF Insert entity and can be placed in any layout: Modelspace, any Paperspace or BlockLayout (which enables nested block references). Every block reference can be scaled and rotated individually.

Let's insert some random flags into the modelspace:

```python
# Get the modelspace of the drawing.
msp = doc.modelspace()

# Get 50 random placing points.
placing_points = [get_random_point() for _ in range(50)]

for point in placing_points:
    # Every flag has a different scaling and a rotation of -15 deg.
    random_scale = 0.5 + random.random() * 2.0
    # Add a block reference to the block named 'FLAG' at the coordinates 'point'.
    msp.add_blockref('FLAG', point, dxfattribs={
        'xscale': random_scale,
        'yscale': random_scale,
        'rotation': -15
    })

# Save the drawing.
doc.saveas("blockref_tutorial.dxf")
```

Query all block references of block FLAG:

```python
for flag_ref in msp.query('INSERT[name=="FLAG"]'):
    print(str(flag_ref))
```

When inserting a block reference into the modelspace or another block layout with different units, the scaling factor between these units should be applied as scaling attributes (xscale, ...) e.g. modelspace in meters and block in centimeters, xscale has to be 0.01.

What are Attributes?

An attribute (Attrib) is a text annotation attached to a block reference with an associated tag. Attributes are often used to add information to blocks which can be evaluated and exported by CAD programs. An attribute can be visible or hidden. The simple way to use attributes is just to add an attribute to a block reference by Insert.add_attrib(), but the attribute is geometrically not related to the block reference, so you have to calculate the insertion point, rotation and scaling of the attribute by yourself.
Using Attribute Definitions

The second way to use attributes in block references is a two step process, first step is to create an attribute definition (template) in the block definition, the second step is adding the block reference by `Layout.add_blockref()` and attach and fill attribute automatically by the `add_auto_attribs()` method to the block reference. The advantage of this method is that all attributes are placed relative to the block base point with the same rotation and scaling as the block, but has the disadvantage that non uniform scaling is not handled very well. The method `Layout.add_auto_blockref()` handles non uniform scaling better by wrapping the block reference and its attributes into an anonymous block and let the CAD application do the transformation work which will create correct graphical representations at least by AutoCAD and BricsCAD. This method has the disadvantage of a more complex evaluation of attached attributes.

Using attribute definitions (`Attdef`):

```python
# Define some attributes for the block 'FLAG', placed relative
to the base point, (0, 0) in this case.
flag.add_attdef('NAME', (0.5, -0.5), dxfattribs={'height': 0.5, 'color': 3})
flag.add_attdef('XPOS', (0.5, -1.0), dxfattribs={'height': 0.25, 'color': 4})
flag.add_attdef('YPOS', (0.5, -1.5), dxfattribs={'height': 0.25, 'color': 4})

# Get another 50 random placing points.
placing_points = [get_random_point() for _ in range(50)]

for number, point in enumerate(placing_points):
    # values is a dict with the attribute tag as item-key and
    # the attribute text content as item-value.
    values = {
        'NAME': 'P(%d)' % (number + 1),
        'XPOS': 'x = %.3f' % point[0],
        'YPOS': 'y = %.3f' % point[1]
    }

    # Every flag has a different scaling and a rotation of +15 deg.
    random_scale = 0.5 + random.random() * 2.0
    blockref = msp.add_blockref('FLAG', point, dxfattribs={
        'rotation': 15
    }).set_scale(random_scale)
    blockref.add_auto_attribs(values)

# Save the drawing.
doc.saveas("auto_blockref_tutorial.dxf")
```

Get/Set Attributes of Existing Block References

See the howto: Get/Set Block Reference Attributes

Evaluate Wrapped Block References

As mentioned above evaluation of block references wrapped into anonymous blocks is complex:

```python
# Collect all anonymous block references starting with '*U'
anonymous_block.refs = modelspace.query('INSERT[name ? "^\*U.+"]')

# Collect real references to 'FLAG'
```

(continues on next page)
Exploding Block References

This is an advanced and still experimental feature and because ezdxf is still not a CAD application, the results may not be perfect. **Non uniform scaling** lead to incorrect results for text entities (TEXT, MTEXT, ATTRIB) and some other entities like HATCH with arc or ellipse path segments.

By default the “exploded” entities are added to the same layout as the block reference is located.

```python
for flag_ref in msp.query('INSERT[name=="FLAG"]':
    flag_ref.explode()
```

Examine Entities of Block References

If you just want to examine the entities of a block reference use the `virtual_entities()` method. This methods yields “virtual” entities with attributes identical to “exploded” entities but they are not stored in the entity database, have no handle and are not assigned to any layout.

```python
for flag_ref in msp.query('INSERT[name=="FLAG"]':
    for entity in flag_ref.virtual_entities():
        if entity.dxftype() == 'LWPOLYLINE':
            print(f'Found {str(entity)}.
```

6.5.5 Tutorial for LWPolyline

The *LWPolyline* is defined as a single graphic entity, which differs from the old-style *Polyline* entity, which is defined as a group of sub-entities. *LWPolyline* display faster (in AutoCAD) and consume less disk space, it is a planar element, therefore all points in *OCS* as \((x, y)\) tuples (*LWPolyline.dxf.elevation* is the z-axis value).

Create a simple polyline:

```python
import ezdxf
doc = ezdxf.new('R2000')
msp = doc.modelspace()
points = [(0, 0), (3, 0), (6, 3), (6, 6)]
msp.add_lwpolyline(points)
doc.saveas("lwpolyline1.dxf")
```
Append multiple points to a polyline:

```python
doc = ezdxf.readfile("lwpolyline1.dxf")
msp = doc.modelspace()
line = msp.query('LWPOLYLINE')[0]  # take first LWPolyline
line.append_points([(8, 7), (10, 7)])
doc.saveas("lwpolyline2.dxf")
```

Getting points always returns a 5-tuple *(x, y, start_width, ent_width, bulge)*, `start_width`, `end_width` and `bulge` is 0 if not present:

```python
first_point = line[0]
x, y, start_width, end_width, bulge = first_point
```

Use context manager to edit polyline points, this method was introduced because accessing single points was very slow, but since `ezdxf` v0.8.9, direct access by index operator `[]` is very fast and using the context manager is not required anymore. Advantage of the context manager is the ability to use a user defined point format:

```python
doc = ezdxf.readfile("lwpolyline2.dxf")
msp = doc.modelspace()
line = msp.query('LWPOLYLINE').first  # take first LWPolyline, 'first' was introduced with v0.10
with line.points('xyseb') as points:
  # points is a standard python list
  # existing points are 5-tuples, but new points can be
  # set as (x, y, [start_width, [end_width, [bulge]]]) tuple
  # set start_width, end_width to 0 to be ignored (x, y, 0, 0, bulge).
  del points[-2:]  # delete last 2 points
  points.extend([(4, 7), (0, 7)])  # adding 2 other points
  # the same as one command
  # points[-2:] = [(4, 7), (0, 7)]
doc.saveas("lwpolyline3.dxf")
```

Each line segment can have a different start- and end-width, if omitted start- and end-width is 0:

```python
doc = ezdxf.new('R2000')
msp = doc.modelspace()

# point format = (x, y, [start_width, [end_width, [bulge]]])
# set start_width, end_width to 0 to be ignored (x, y, 0, 0, bulge).
points = [(0, 0, .1, .15), (3, 0, .2, .25), (6, 3, .3, .35), (6, 6)]
msp.add_lwpolyline(points)
doc.saveas("lwpolyline4.dxf")
```

The first point carries the start- and end-width of the first segment, the second point of the second segment and so on, the start- and end-width value of the last point is used for the closing segment if polyline is closed else the values are ignored. Start- and end-width only works if the DXF attribute `dxf.const_width` is unset, to be sure delete it:

```python
del line.dxf.const_width  # no exception will be raised if const_width is already unset.
```
LWPolyline can also have curved elements, they are defined by the *Bulge value*:

```python
doc = ezdxf.new('R2000')
msp = doc.modelspace()

# point format = (x, y, [start_width, [end_width, [bulge]]])
# set start_width, end_width to 0 to be ignored (x, y, 0, 0, bulge).
points = [(0, 0, 0, .05), (3, 0, .1, .2, -.5), (6, 0, .1, .05), (9, 0)]
msp.add_lwpolyline(points)
doc.saveas("lwpolyline5.dxf")
```

The curved segment is drawn from the point which defines the bulge value to the following point, the curved segment is always an arc. The bulge value defines the ratio of the arc sagitta (segment height \( h \)) to half line segment length (point distance), a bulge value of 1 defines a semicircle. \( \text{bulge} > 0 \) the curve is on the right side of the vertex connection line, \( \text{bulge} < 0 \) the curve is on the left side.

**ezdxf** v0.8.9 supports a user defined points format, default is *xyseb*:

- \( x = \) x coordinate
- \( y = \) y coordinate
- \( s = \) start width
- \( e = \) end width
- \( b = \) bulge value
- \( v = (x, y) \) as tuple

```python
msp.add_lwpolyline([(0, 0, 0), (10, 0, 1), (20, 0, 0)], format='xyseb')
msp.add_lwpolyline([(0, 10, 0), (10, 10, .5), (20, 10, 0)], format='xyb')
```
6.5.6 Tutorial for Text

Add a simple one line text entity by factory function `add_text()`.

```python
import ezdx

# TEXT is a basic entity and is supported by every DXF version.
# Argument setup=True for adding standard linetypes and text styles.
doc = ezdx.new('R12', setup=True)
msp = doc.modelspace()

# use set_pos() for proper TEXT alignment:
# The relations between DXF attributes 'halign', 'valign',
# 'insert' and 'align_point' are tricky.
msp.add_text("A Simple Text").set_pos((2, 3), align='MIDDLE_RIGHT')

# Using a text style
msp.add_text("Text Style Example: Liberation Serif",
    dxfattribs={
        'style': 'LiberationSerif',
        'height': 0.35
    }).set_pos((2, 6), align='LEFT')
```

(continues on next page)
doc.saveas("simple_text.dxf")

Valid text alignments for argument `align` in `Text.set_pos()`:

<table>
<thead>
<tr>
<th>Vert/Horiz</th>
<th>Left</th>
<th>Center</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top</td>
<td>TOP_LEFT</td>
<td>TOP_CENTER</td>
<td>TOP_RIGHT</td>
</tr>
<tr>
<td>Middle</td>
<td>MIDDLE_LEFT</td>
<td>MIDDLE_CENTER</td>
<td>MIDDLE_RIGHT</td>
</tr>
<tr>
<td>Bottom</td>
<td>BOTTOM_LEFT</td>
<td>BOTTOM_CENTER</td>
<td>BOTTOM_RIGHT</td>
</tr>
<tr>
<td>Baseline</td>
<td>LEFT</td>
<td>CENTER</td>
<td>RIGHT</td>
</tr>
</tbody>
</table>

Special alignments are ALIGNED and FIT, they require a second alignment point, the text is justified with the vertical alignment `Baseline` on the virtual line between these two points.

<table>
<thead>
<tr>
<th>Alignment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIGNED</td>
<td>Text is stretched or compressed to fit exactly between <code>p1</code> and <code>p2</code> and the text height is also adjusted to preserve height/width ratio.</td>
</tr>
<tr>
<td>FIT</td>
<td>Text is stretched or compressed to fit exactly between <code>p1</code> and <code>p2</code> but only the text width is adjusted, the text height is fixed by the <code>height</code> attribute.</td>
</tr>
<tr>
<td>MIDDLE</td>
<td>also a special adjustment, but the result is the same as for MIDDLE_CENTER.</td>
</tr>
</tbody>
</table>

**Standard Text Styles**

Setup some standard text styles and linetypes by argument `setup=True`:

```
doc = ezdxf.new('R12', setup=True)
```

Replaced all proprietary font declarations in `setup_styles()` (ARIAL, ARIAL_NARROW, ISOCPEUR and TIMES) by open source fonts, this is also the style name (e.g. `{'style': 'OpenSans-Italic'}`):
New Text Style

Creating a new text style is simple:

```python
doc.styles.new('myStandard', dxfattribs={'font' : 'OpenSans-Regular.ttf'})
```

But getting the correct font name is often not that simple, especially on Windows. This shows the required steps to get the font name for *Open Sans*:

- open font folder `c:\windows\fonts`
- select and open the font-family *Open Sans*
- right-click on *Open Sans Standard* and select *Properties*
- on top of the first tab you see the font name: 'OpenSans-Regular.ttf'

The style name has to be unique in the DXF document, else `ezdxf` will raise a `DXFTableEntryError` exception. To replace an existing entry, delete the existing entry by `doc.styles.remove(name)`, and add the replacement entry.

3D Text

It is possible to place the 2D *Text* entity into 3D space by using the *OCS*, for further information see: *Tutorial for OCS/UCS Usage*.

6.5.7 Tutorial for MText

The *MText* entity is a multi line entity with extended formatting possibilities and requires at least DXF version R2000, to use all features (e.g. background fill) DXF R2007 is required.

Prolog code:

```python
import ezdxf

doc = ezdxf.new('R2007', setup=True)
msp = doc.modelspace()

lorem_ipsum = ""
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
"
```

Adding a MText entity

The MText entity can be added to any layout (modelspace, paperspace or block) by the `add_mtext()` function.

```python
# store MText entity for additional manipulations
mtext = msp.add_mtext(lorem_ipsum, dxfattribs={'style': 'OpenSans'})
```
This adds a MText entity with text style 'OpenSans'. The MText content can be accessed by the text attribute, this attribute can be edited like any Python string:

```python
mtext.text += 'Append additional text to the MText entity.'
# even shorter with __iadd__() support:
mtext += 'Append additional text to the MText entity.'
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Append additional text to the MText entity.

**Important:** Line endings `\n` will be replaced by the MTEXT line endings `\P` at DXF export, but **not** vice versa `\P` by `\n` at DXF file loading.

### Text placement

The location of the MText entity is defined by the MText.dxf.insert and the MText.dxf.attachment_point attributes. The attachment_point defines the text alignment relative to the insert location, default value is 1.

Attachment point constants defined in ezdxf.lldxf.const:

<table>
<thead>
<tr>
<th>MText.dxf.attachment_point</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTEXT_TOP_LEFT</td>
<td>1</td>
</tr>
<tr>
<td>MTEXT_TOP_CENTER</td>
<td>2</td>
</tr>
<tr>
<td>MTEXT_TOP_RIGHT</td>
<td>3</td>
</tr>
<tr>
<td>MTEXT_MIDDLE_LEFT</td>
<td>4</td>
</tr>
<tr>
<td>MTEXT_MIDDLE_CENTER</td>
<td>5</td>
</tr>
<tr>
<td>MTEXT_MIDDLE_RIGHT</td>
<td>6</td>
</tr>
<tr>
<td>MTEXT_BOTTOM_LEFT</td>
<td>7</td>
</tr>
<tr>
<td>MTEXT_BOTTOM_CENTER</td>
<td>8</td>
</tr>
<tr>
<td>MTEXT_BOTTOM_RIGHT</td>
<td>9</td>
</tr>
</tbody>
</table>

The MText entity has a method for setting insert, attachment_point and rotation attributes by one call: `set_location()`

### Character height

The character height is defined by the DXF attribute MText.dxf.char_height in drawing units, which has also consequences for the line spacing of the MText entity:
mtext.dxf.char_height = 0.5

The character height can be changed inline, see also MText formatting and MText Inline Codes.

Text rotation (direction)

The MText.dxf.rotation attribute defines the text rotation as angle between the x-axis and the horizontal direction of the text in degrees. The MText.dxf.text_direction attribute defines the horizontal direction of MText as vector in WCS or OCS, if an OCS is defined. Both attributes can be present at the same entity, in this case the MText.dxf.text_direction attribute has the higher priority.

The MText entity has two methods to get/set rotation: get_rotation() returns the rotation angle in degrees independent from definition as angle or direction, and set_rotation() set the rotation attribute and removes the text_direction attribute if present.

Defining a wrapping border

The wrapping border limits the text width and forces a line break for text beyond this border. Without attribute dxf.width (or setting 0) the lines are wrapped only at the regular line endings \P or \n, setting the reference column width forces additional line wrappings at the given width. The text height can not be limited, the text always occupies as much space as needed.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Append additional text to the MText entity.

MText formatting

MText supports inline formatting by special codes: MText Inline Codes

mtext.text = "{\C1red text} - {\C3green text} - {\C5blue text}"
Stacked text

MText also supports stacked text:

```plaintext
# the space '' in front of 'Lower' and the ';' behind 'Lower' are necessary
# combined with vertical center alignment
mtext.text = "\\A1\\Upper^ Lower; - \SUpper/ Lower;} - \SUpper# Lower;"
```

Available helper function for text formatting:

- `set_color()` - append text color change
- `set_font()` - append text font change
- `add_stacked_text()` - append stacked text

Background color (filling)

The MText entity can have a background filling:

- `AutoCAD Color Index (ACI)`
- true color value as (r, g, b) tuple
- color name as string, use special name `canvas` to use the canvas background color

Because of the complex dependencies ezdx

```python
mtext.set_bg_color(2, scale=1.5)
```

The parameter `scale` determines how much border there is around the text, the value is based on the text height, and should be in the range of 1 - 5, where 1 fits exact the MText entity.
6.5.8 Tutorial for Spline

Background information about B-spline at Wikipedia.

Splines from fit points

Splines can be defined by fit points only, this means the curve goes through all given fit points. AutoCAD and BricsCAD generates required control points and knot values by itself, if only fit points are present.

Create a simple spline:

```python
doc = ezdxf.new('R2000')
fit_points = [(0, 0, 0), (750, 500, 0), (1750, 500, 0), (2250, 1250, 0)]
msp = doc.modelspace()
spline = msp.add_spline(fit_points)
```
Append a fit point to a spline:

```python
# fit_points, control_points, knots and weights are list-like containers:
spline.fit_points.append((2250, 2500, 0))
```
You can set additional control points, but if they do not fit the auto-generated AutoCAD values, they will be ignored and don’t mess around with knot values.

Solve problems of incorrect values after editing a spline generated by AutoCAD:

```python
doc = ezdxf.readfile("AutoCAD_generated.dxf")
msp = doc.modelspace()
spline = msp.query('SPLINE').first

# fit_points, control_points, knots and weights are list-like objects:
spline.fit_points.append((2250, 2500, 0))
```

As far as I have tested, this approach works without complaints from AutoCAD, but for the case of problems remove invalid data:

```python
# current control points do not match spline defined by fit points
spline.control_points = []

# count of knots is not correct:
# count of knots = count of control points + degree + 1
spline.knots = []

# same for weights, count of weights == count of control points
spline.weights = []
```

**Splines by control points**

To create splines from fit points is the easiest way to create splines, but this method is also the least accurate, because a spline is defined by control points and knot values, which are generated for the case of a definition by fit points, and the worst fact is that for every given set of fit points exist an infinite number of possible splines as solution.

AutoCAD (and BricsCAD also) uses an proprietary algorithm to generate control points and knot values from fit points, which differs from the well documented Global Curve Interpolation. Therefore splines generated from fit points by `ezdxf` do not match splines generated by AutoCAD (BricsCAD).

To ensure the same spline geometry for all CAD applications, the spline has to be defined by control points. The method `add_spline_control_frame()` adds a spline trough fit points by calculating the control points by the Global Curve Interpolation algorithm. There is also a low level function `ezdxf.math.global_bspline_interpolation()` which calculates the control points from fit points.

```python
msp.add_spline_control_frame(fit_points, method='uniform', dxfattribs={'color': 1})
msp.add_spline_control_frame(fit_points, method='chord', dxfattribs={'color': 3})
msp.add_spline_control_frame(fit_points, method='centripetal', dxfattribs={'color': 5})
```

- black curve: AutoCAD/BricsCAD spline generated from fit points
- red curve: spline curve interpolation, “uniform” method
- green curve: spline curve interpolation, “chord” method
- blue curve: spline curve interpolation, “centripetal” method
Open Spline

Add and open (clamped) spline defined by control points with the method `add_open_spline()`. If no knot values are given, an open uniform knot vector will be generated. A clamped B-spline starts at the first control point and ends at the last control point.

```python
control_points = [(0, 0, 0), (1250, 1560, 0), (3130, 610, 0), (2250, 1250, 0)]
msp.add_open_spline(control_points)
```
Rational Spline

Rational B-splines have a weight for every control point, which can raise or lower the influence of the control point, default weight = 1, to lower the influence set a weight < 1 to raise the influence set a weight > 1. The count of weights has to be always equal to the count of control points.

Example to raise the influence of the first control point:

```python
msp.add_closed_rational_spline(control_points, weights=[3, 1, 1, 1])
```
Spline properties

Check if spline is a closed curve or close/open spline, for a closed spline the last point is connected to the first point:

```python
if spline.closed:
    # this spline is closed
    pass

# close spline
spline.closed = True

# open spline
spline.closed = False
```

Set start- and end tangent for splines defined by fit points:

```python
spline.dxf.start_tangent = (0, 1, 0)  # in y-axis
spline.dxf.end_tangent = (1, 0, 0)   # in x-axis
```

Get data count as stored in DXF file:

```python
count = spline.dxf.n_fit_points
count = spline.dxf.n_control_points
count = spline.dxf.n_knots
```

Get data count of real existing data:

```python
count = spline.fit_point_count
count = spline.control_point_count
count = spline.knot_count
```

6.5.9 Tutorial for Polyface

coming soon...

6.5.10 Tutorial for Mesh

Create a cube mesh by direct access to base data structures:

```python
import ezdxf

define

# 8 corner vertices
cube_vertices = [
    (0, 0, 0),
    (1, 0, 0),
    (1, 1, 0),
    (0, 1, 0),
    (0, 0, 1),
    (1, 0, 1),
    (1, 1, 1),
    (0, 1, 1),
]

# 6 cube faces
```
cube_faces = [
    [0, 1, 2, 3],
    [4, 5, 6, 7],
    [0, 1, 5, 4],
    [1, 2, 6, 5],
    [3, 2, 6, 7],
    [0, 3, 7, 4]
]

doc = ezdxf.new('R2000')  # MESH requires DXF R2000 or later
msp = doc.modelspace()
(mesh = msp.add_mesh())
mesh.dxf.subdivision_levels = 0  # do not subdivide cube, 0 is the default value
with mesh.edit_data() as mesh_data:
    mesh_data.vertices = cube_vertices
    mesh_data.faces = cube_faces

doc.saveas("cube_mesh_1.dxf")

Create a cube mesh by method calls:

```python
import ezdxf

# 8 corner vertices
p = [
    (0, 0, 0),
    (1, 0, 0),
    (1, 1, 0),
    (0, 1, 0),
    (0, 0, 1),
    (1, 0, 1),
    (1, 1, 1),
    (0, 1, 1),
]

doc = ezdxf.new('R2000')  # MESH requires DXF R2000 or later
msp = doc.modelspace()
(mesh = msp.add_mesh())
with mesh.edit_data() as mesh_data:
    mesh_data.add_face([p[0], p[1], p[2], p[3]])
    mesh_data.add_face([p[4], p[5], p[6], p[7]])
    mesh_data.add_face([p[0], p[1], p[5], p[4]])
    mesh_data.add_face([p[1], p[2], p[6], p[5]])
    mesh_data.add_face([p[3], p[2], p[6], p[7]])
    mesh_data.add_face([p[0], p[3], p[7], p[4]])
    mesh_data.optimize()  # optional, minimizes vertex count

doc.saveas("cube_mesh_2.dxf")
```

### 6.5.11 Tutorial for Hatch

---

6.5. Tutorials
Create hatches with one boundary path

The simplest form of the Hatch entity has one polyline path with only straight lines as boundary path:

```python
import ezdxf
doc = ezdx.new('R2000')  # hatch requires the DXF R2000 (AC1015) format or later
msp = doc.modelspace()  # adding entities to the model space

hatch = msp.add_hatch(color=2)  # by default a solid fill hatch with fill color=7
  # (white/black)
# every boundary path is always a 2D element
# vertex format for the polyline path is: (x, y[, bulge])
# there are no bulge values in this example
hatch.paths.add_polyline_path([(0, 0), (10, 0), (10, 10), (0, 10)], is_closed=True)

doc.saveas("solid_hatch_polyline_path.dxf")
```

But like all polyline entities the polyline path can also have bulge values:

```python
import ezdxf
doc = ezdx.new('R2000')  # hatch requires the DXF R2000 (AC1015) format or later
msp = doc.modelspace()  # adding entities to the model space

hatch = msp.add_hatch(color=2)  # by default a solid fill hatch with fill color=7
  # (white/black)
# every boundary path is always a 2D element
# vertex format for the polyline path is: (x, y[, bulge])
# bulge value 1 = an arc with diameter=10 (= distance to next vertex * bulge value)
# bulge value > 0 ... arc is right of line
# bulge value < 0 ... arc is left of line
hatch.paths.add_polyline_path([(0, 0, 1), (10, 0), (10, 10, -0.5), (0, 10)], is_closed=True)

doc.saveas("solid_hatch_polyline_path_with_bulge.dxf")
```

The most flexible way to define a boundary path is the edge path. An edge path consist of a number of edges and each edge can be one of the following elements:

- `line` `EdgePath.add_line()`
- `arc` `EdgePath.add_arc()`
- `ellipse` `EdgePath.add_ellipse()`
- `spline` `EdgePath.add_spline()`

Create a solid hatch with an edge path (ellipse) as boundary path:

```python
import ezdxf
doc = ezdx.new('R2000')  # hatch requires the DXF R2000 (AC1015) format or later
msp = doc.modelspace()  # adding entities to the model space

# important: major axis >= minor axis (ratio <= 1.)
# minor axis length = major axis length * ratio
```

(continues on next page)
msp.add_ellipse((0, 0), major_axis=(0, 10), ratio=0.5)

# by default a solid fill hatch with fill color=7 (white/black)
hatch = msp.add_hatch(color=2)

# every boundary path is always a 2D element
edge_path = hatch.paths.add_edge_path()
# each edge path can contain line arc, ellipse and spline elements
# important: major axis >= minor axis (ratio <= 1.)
edge_path.add_ellipse((0, 0), major_axis=(0, 10), ratio=0.5)

doc.saveas("solid_hatch_ellipse.dxf")

Create hatches with multiple boundary paths (islands)

The DXF attribute `hatch_style` defines the island detection style:

<table>
<thead>
<tr>
<th></th>
<th>nested - altering filled and unfilled areas</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>outer - area between external and outermost path is filled</td>
</tr>
<tr>
<td>2</td>
<td>ignore - external path is filled</td>
</tr>
</tbody>
</table>

hatch = msp.add_hatch(color=1, dxfattribs={
    'hatch_style': ezdxf.const.HATCH_STYLE_NESTED,
    # 0 = nested: ezdxf.const.HATCH_STYLE_NESTED
    # 1 = outer: ezdxf.const.HATCH_STYLE_OUTERMOST
    # 2 = ignore: ezdxf.const.HATCH_STYLE_IGNORE
})

# The first path has to set flag: 1 = external
# flag const.BOUNDARY_PATH_POLYLINE is added (OR) automatically
hatch.paths.add_polyline_path(
    [(0, 0), (10, 0), (10, 10), (0, 10)], is_closed=True,
    flags=ezdxf.const.BOUNDARY_PATH_EXTERNAL)

This is also the result for all 4 paths and `hatch_style` set to 2 (ignore).

# The second path has to set flag: 16 = outermost
hatch.paths.add_polyline_path(
    [(1, 1), (9, 1), (9, 9), (1, 9)], is_closed=True,
    flags=ezdxf.const.BOUNDARY_PATH_OUTERMOST)
This is also the result for all 4 paths and `hatch_style` set to 1 (outer).

```python
# The third path has to set flag: 0 = default
hatch.paths.add_polyline_path(
    [(2, 2), (8, 2), (8, 8), (2, 8)],
    is_closed=True,
    flags=ezdxf.const.BOUNDARY_PATH_DEFAULT)
```

```
# The forth path has to set flag: 0 = default, and so on
hatch.paths.add_polyline_path(
    [(3, 3), (7, 3), (7, 7), (3, 7)],
    is_closed=True,
    flags=ezdxf.const.BOUNDARY_PATH_DEFAULT)
```

The expected result of combinations of various `hatch_style` values and paths `flags`, or the handling of overlapping paths is not documented by the DXF reference, so don’t ask me, ask Autodesk or just try it by yourself and post your experience in the forum.
Example for Edge Path Boundary

```python
hatch = msp.add_hatch(color=1)

# 1. polyline path
hatch.paths.add_polyline_path(
    [240, 210, 0],
    [0, 210, 0],
    [0, 0, 0.],
    [240, 0, 0],
    is_closed=1,
    flags=ezdxf.const.BOUNDARY_PATH_EXTERNAL,
)

# 2. edge path
edge_path = hatch.paths.add_edge_path(flags=ezdxf.const.BOUNDARY_PATH_OUTERMOST)
edge_path.add_spline(
    control_points=[
        (126.658105895725, 177.0823706957212),
        (141.5497003747484, 187.8907860433995),
        (205.8997365206943, 154.7946313459515),
        (113.0168862297068, 117.8189380884978),
        (202.9816918983783, 63.17222935389572),
        (157.363511042264, 26.4621294342132),
        (144.8204003260554, 28.4383294369643)
    ],
    knot_values=[
        0.0, 0.0, 0.0, 0.0, 55.20174685732758, 98.33239645153571,
        175.1126541251052, 213.206156683142, 213.206156683142,
        213.206156683142
    ],
)
edge_path.add_arc(
    center=(152.6378550678883, 128.3209356351659),
    radius=100.1880612627354,
    start_angle=94.4752130054052,
    end_angle=177.1345242028005,
)
edge_path.add_line(
    [52.57506282464041, 123.3124200796114],
    [240, 210, 0],
    [0, 210, 0],
    [0, 0, 0.],
    [240, 0, 0],
    is_closed=1,
    flags=ezdxf.const.BOUNDARY_PATH_EXTERNAL,
)
```

6.5. Tutorials
Associative Boundary Paths

A HATCH entity can be associative to a base geometry, which means if the base geometry is edited in a CAD application the HATCH get the same modification. Because ezdxf is not a CAD application, this association is not maintained nor verified by ezdxf, so if you modify the base geometry afterwards the geometry of the boundary path is not updated and no verification is done to check if the associated geometry matches the boundary path, this opens many possibilities to create invalid DXF files: USE WITH CARE.

This example associates a LWPOLYLINE entity to the hatch created from the LWPOLYLINE vertices:

```python
# Create base geometry
lwpolyline = msp.add_lwpolyline(
    [(0, 0, 0), (10, 0, .5), (10, 10, 0), (0, 10, 0)],
    format='xyb',
    close=True,
)

hatch = msp.add_hatch(color=1)
path = hatch.paths.add_polyline_path(
    # get path vertices from associated LWPOLYLINE entity
    lwpolyline.get_points(format='xyb'),
    # get closed state also from associated LWPOLYLINE entity
    is_closed=lwpolyline.closed,
)

# Set association between boundary path and LWPOLYLINE
hatch.associate(path, [lwpolyline])
```

An EdgePath needs associations to all geometry entities forming the boundary path.
Predefined Hatch Pattern

Use predefined hatch pattern by name:

```python
hatch.set_pattern_fill('ANSI31', scale=0.5)
```
Create hatches with gradient fill

TODO

### 6.5.12 Tutorial for Hatch Pattern Definition

TODO

### 6.5.13 Tutorial for Image and ImageDef

Insert a raster image into a DXF drawing, the raster image is NOT embedded into the DXF file:

```python
import ezdxf

doc = ezdxf.new('AC1015')  # image requires the DXF R2000 format or later
my_image_def = doc.add_image_def(filename='mycat.jpg', size_in_pixel=(640, 360))
# The IMAGEDEF entity is like a block definition, it just defines the image
msp = doc.modelspace()
# add first image
msp.add_image(insert=(2, 1), size_in_units=(6.4, 3.6), image_def=my_image_def,
              rotation=0)
# The IMAGE entity is like the INSERT entity, it creates an image reference,
# and there can be multiple references to the same picture in a drawing.
msp.add_image(insert=(4, 5), size_in_units=(3.2, 1.8), image_def=my_image_def,
              rotation=30)

# get existing image definitions, Important: IMAGEDEFS resides in the objects section
image_defs = doc.objects.query('IMAGEDEF')  # get all image defs in drawing

doc.saveas("dxf_with_cat.dxf")
```

### 6.5.14 Tutorial for Underlay and UnderlayDefinition

Insert a PDF, DWF, DWFx or DGN file as drawing underlay, the underlay file is NOT embedded into the DXF file:

```python
import ezdxf

doc = ezdxf.new('AC1015')  # underlay requires the DXF R2000 format or later
my_underlay_def = doc.add_underlay_def(filename='my_underlay.pdf', name='1')
# The (PDF)DEFINITION entity is like a block definition, it just defines the underlay
# 'name' is misleading, because it defines the page/sheet to be displayed
# PDF: name is the page number to display
# DGN: name='default' ???
# DWF: ???
msp = doc.modelspace()
# add first underlay
msp.add_underlay(my_underlay_def, insert=(2, 1, 0), scale=0.05)
# The (PDF)UNDERLAY entity is like the INSERT entity, it creates an underlay reference,
```

(continues on next page)
ezdxf Documentation, Release 0.16.2

(continued from previous page)

# and there can be multiple references to the same underlay in a drawing.
msp.add_underlay(my_underlay_def, insert=(4, 5, 0), scale=.5, rotation=30)
# get existing underlay definitions, Important: UNDERLAYDEFs resides in the objects
˓→section
pdf_defs = doc.objects.query('PDFDEFINITION') # get all pdf underlay defs in drawing
doc.saveas("dxf_with_underlay.dxf")

6.5.15 Tutorial for Linetypes
Simple line type example:

You can define your own line types. A DXF linetype definition consists of name, description and elements:
elements = [total_pattern_length, elem1, elem2, ...]

total_pattern_length Sum of all linetype elements (absolute vaues)
elem if elem > 0 it is a line, if elem < 0 it is gap, if elem == 0.0 it is a dot
Create a new linetype definition:
import ezdxf
from ezdxf.tools.standards import linetypes

# some predefined line types

doc = ezdxf.new()
msp = modelspace()
my_line_types = [
("DOTTED", "Dotted . . . . . . . . . . . . . . . .", [0.2, 0.0, -0.
˓→2]),
("DOTTEDX2", "Dotted (2x) .
.
.
.
.
.
.
. ", [0.4, 0.0, -0.
˓→4]),
("DOTTED2", "Dotted (.5) . . . . . . . . . . . . . . . . . . . ", [0.1, 0.0, -0.
˓→1]),
]
for name, desc, pattern in my_line_types:
if name not in doc.linetypes:
doc.linetypes.new(name=name, dxfattribs={'description': desc, 'pattern':
˓→pattern})

Setup some predefined linetypes:
for name, desc, pattern in linetypes():
if name not in doc.linetypes:
doc.linetypes.new(name=name, dxfattribs={'description': desc, 'pattern':
˓→pattern})

Check Available Linetypes
The linetypes object supports some standard Python protocols:
66

Chapter 6. Contents


# iteration
print('available line types:')
for linetype in doc.linetypes:
    print('{}, {}'.format(linetype.dxf.name, linetype.dxf.description))

# check for existing line type
if 'DOTTED' in doc.linetypes:
    pass

count = len(doc.linetypes)  # total count of linetypes

Removing Linetypes

**Warning:** Deleting of linetypes still in use generates invalid DXF files.

You can delete a linetype:

doc.layers.remove('DASHED')

This just deletes the linetype definition, all DXF entity with the DXF attribute linetype set to DASHED still refers to linetype DASHED and AutoCAD will not open DXF files with undefined line types.

### 6.5.16 Tutorial for Complex Linetypes

In DXF R13 Autodesk introduced complex line types, containing TEXT or SHAPES in line types. *ezdxf* v0.8.4 and later supports complex line types.

Complex line type example with text:

```
GAS ----GAS----GAS----GAS----GAS----GAS--
```

Complex line type example with shapes:

```
□□□□□□□□□□
```

For simplicity the pattern string for complex line types is mostly the same string as the pattern definition strings in AutoCAD .lin files.

Example for complex line type TEXT:

```python
doc = ezdxf.new('R2018')  # DXF R13 or later is required
doc.linetypes.new('GASLEITUNG2', dxfattribs={
    'description': 'Gasleitung2 ----GAS----GAS----GAS----GAS----GAS----GAS--',
    'length': 1,  # required for complex line types
    # line type definition in acadlt.lin:
    'pattern': 'A,.5,-.2,["GAS",STANDARD,S=.1,U=0.0,X=-0.1,Y=-.05],-.25',
})
```

The pattern always starts with an `A`, the following float values have the same meaning as for simple line types, a value > 0 is a line, a value < 0 is a gap, and a 0 is a point, the `[]` starts the complex part of the line pattern. A following text in quotes defines a TEXT type, a following text without quotes defines a SHAPE type, in .lin files the shape type is a shape name, but *ezdxf* can not translate this name into the required shape file index, so **YOU** have to translate this
name into the shape file index (e.g. saving the file with AutoCAD as DXF and searching for the line type definition, see also DXF Internals: \textit{LTYPE Table}).

The second parameter is the text style for a TEXT type and the shape file name for the SHAPE type, the shape file has to be in the same directory as the DXF file. The following parameters in the scheme of \( S=1.0 \) are:

\begin{itemize}
  \item \( S \ldots \) scaling factor, always \( > 0 \), if \( S=0 \) the TEXT or SHAPE is not visible
  \item \( R \) or \( U \ldots \) rotation relative to the line direction
  \item \( X \ldots \) x direction offset (along the line)
  \item \( Y \ldots \) y direction offset (perpendicular to the line)
\end{itemize}

The parameters are case insensitive. \texttt{]} ends the complex part of the line pattern.

The fine tuning of this parameters is still a try an error process for me, for TEXT the scaling factor (STANDARD text style) sets the text height (\( S=.1 \) the text is .1 units in height), by shifting in \( y \) direction by half of the scaling factor, the center of the text is on the line. For the \( x \) direction it seems to be a good practice to place a gap in front of the text and after the text, find \( x \) shifting value and gap sizes by try and error. The overall length is at least the sum of all line and gap definitions (absolute values).

Example for complex line type SHAPE:

\begin{verbatim}
doc.linetypes.new('GRENZE2', dxfattribs={
    'description': 'Grenze eckig ----[]-----[]----[]-----[]----[]--',
    'length': 1.45,  # required for complex line types
    # line type definition in acadlt.lin:
    # A,.25,-.1,[BOX,ltypeshp.shx,x=-.1,s=.1],-.1,1
    # replacing BOX by shape index 132 (got index from an AutoCAD file),
    # ezdxf can't get shape index from ltypeshp.shx
    # pattern': 'A,.25,-.1,[132,ltypeshp.shx,x=-.1,s=.1],-.1,1',
})
\end{verbatim}

Complex line types with shapes only work if the associated shape file (ltypeshp.shx) and the DXF file are in the same directory.

\subsection*{6.5.17 Tutorial for OCS/UCS Usage}

For OCS/UCS usage is a basic understanding of vectors required, for a brush up, watch the YouTube tutorials of 3Blue1Brown about Linear Algebra.

Second read the \textit{Coordinate Systems} introduction please.

\textbf{See also:}

The free online book \textit{3D Math Primer for Graphics and Game Development} is a very good resource for learning vector math and other graphic related topics, it is easy to read for beginners and especially targeted to programmers.

For \textit{WCS} there is not much to say as, it is what it is: the main world coordinate system, and a drawing unit can have any real world unit you want. Autodesk added some mechanism to define a scale for dimension and text entities, but because I am not an AutoCAD user, I am not familiar with it, and further more I think this is more an AutoCAD topic than a DXF topic.

\textbf{Object Coordinate System (OCS)}

The \textit{OCS} is used to place planar 2D entities in 3D space. \textbf{ALL} points of a planar entity lay in the same plane, this is also true if the plane is located in 3D space by an OCS. There are three basic DXF attributes that gives a 2D entity its spatial form.
Extrusion

The extrusion vector defines the OCS, it is a normal vector to the base plane of a planar entity. This base plane is always located in the origin of the WCS. But there are some entities like Ellipse, which have an extrusion vector, but do not establish an OCS. For this entities the extrusion vector defines only the extrusion direction and thickness defines the extrusion distance, but all other points in WCS.

Elevation

The elevation value defines the z-axis value for all points of a planar entity, this is an OCS value, and defines the distance of the entity plane from the base plane.

This value exists only in output from DXF versions prior to R11 as separated DXF attribute (group code 38). In DXF R12 and later, the elevation value is supplied as z-axis value of each point. But as always in DXF, this simple rule does not apply to all entities: LWPolyline and Hatch have an DXF attribute elevation, where the z-axis of this point is the elevation height and the x-axis = y-axis = 0.

Thickness

Defines the extrusion distance for an entity.

Note: There is a new edition of this tutorial using UCS based transformation, which are available in ezdxf v0.11 and later: Tutorial for UCS Based Transformations

This edition shows the hard way to accomplish the transformations by low level operations.

Placing 2D Circle in 3D Space

The colors for axis follow the AutoCAD standard:

- red is x-axis
- green is y-axis
- blue is z-axis

```python
import ezdxf
from ezdxf.math import OCS

doc = ezdxf.new('R2010')
msp = doc.modelspace()

# For this example the OCS is rotated around x-axis about 45 degree
# OCS z-axis: x=0, y=1, z=1
# extrusion vector must not normalized here
ocs = OCS((0, 1, 1))
msp.add_circle(
    center=ocs.from_wcs((0, 2, 2)),
    radius=1,
    dxfattribs={
```
(continues on next page)
The following image shows the 2D circle in 3D space in AutoCAD Left and Front view. The blue line shows the OCS z-axis (extrusion direction), elevation is the distance from the origin to the center of the circle in this case 2.828, and you see that the x- and y-axis of OCS and WCS are not aligned.
Placing LWPolyline in 3D Space

For simplicity of calculation I use the UCS class in this example to place a 2D pentagon in 3D space.

```python
calculation_center = (0, 2, 2), # center of pentagon
ux=(1, 0, 0), # x-axis parallel to WCS x-axis
uz=(0, 1, 1), # z-axis

# calculating corner points in local (UCS) coordinates
points = [Vec3.from_deg_angle((360 / 5) * n) for n in range(5)]
# converting UCS into OCS coordinates
ocs_points = list(ucs.points_to_ocs(points))

# LWPOLYLINE accepts only 2D points and has an separated DXF attribute elevation.
# All points have the same z-axis (elevation) in OCS!
elevation = ocs_points[0].z

msp.add_lwpolyline(
    points=ocs_points,
    format='xy', # ignore z-axis
    close=True,
    dxfattribs={
        'elevation': elevation,
        'extrusion': ucs.uz,
        'color': 1,
    }
)
```

The following image shows the 2D pentagon in 3D space in AutoCAD Left, Front and Top view. The three lines from the center of the pentagon show the UCS, the three colored lines in the origin show the OCS the white lines in the origin show the WCS.

The z-axis of the UCS and the OCS show the same direction (extrusion direction), and the x-axis of the UCS and the WCS show the same direction. The elevation is the distance from the origin to the center of the pentagon and all points of the pentagon have the same elevation, and you see that the y-axis of UCS, OCS and WCS are not aligned.
Using UCS to Place 3D Polyline

It is much simpler to use a 3D Polyline to create the 3D pentagon. The UCS class is handy for this example and all kind of 3D operations.

```python
# Using an UCS simplifies 3D operations, but UCS definition can happen later
# calculating corner points in local (UCS) coordinates without Vec3 class
angle = math.radians(360 / 5)
corners_ucs = [(math.cos(angle * n), math.sin(angle * n), 0) for n in range(5)]

# let's do some transformations
tmatrix = Matrix44.chain(  # creating a transformation matrix
    Matrix44.z_rotate(math.radians(15)),  # 1. rotation around z-axis
    Matrix44.translate(0, .333, .333),  # 2. translation
)```

(continues on next page)
transformed_corners_ucs = tmatrix.transform_vertices(corners_ucs)

# transform UCS into WCS
ucs = UCS(
    origin=(0, 2, 2),  # center of pentagon
    ux=(1, 0, 0),  # x-axis parallel to WCS x-axis
    uz=(0, 1, 1),  # z-axis
)
corners_wcs = list(ucs.points_to_wcs(transformed_corners_ucs))

msp.add_polyline3d(
    points=corners_wcs,
    close=True,
)

# add lines from center to corners
center_wcs = ucs.to_wcs((0, .333, .333))
for corner in corners_wcs:
    msp.add_line(center_wcs, corner, dxfattribs={'color': 1})

ucs.render_axis(msp)

Placing 2D Text in 3D Space

The problem by placing text in 3D space is the text rotation, which is always counter clockwise around the OCS z-axis, and 0 degree is in direction of the positive OCS x-axis, and the OCS x-axis is calculated by the Arbitrary Axis Algorithm.

Calculate the OCS rotation angle by converting the TEXT rotation angle (in UCS or WCS) into a vector or begin with text direction as vector, transform this direction vector into OCS and convert the OCS vector back into an angle in the OCS xy-plane (see example), this procedure is available as `UCS.to_ocs_angle_deg() or UCS.to_ocs_angle_rad()`.

AutoCAD supports thickness for the TEXT entity only for .shx fonts and not for true type fonts.
# Thickness for text works only with shx fonts not with true type fonts

doc.styles.new('TXT', dxfattribs={'font': 'romans.shx'})

calculate text direction as angle in OCS:

# convert text rotation in degree into a vector in UCS

text_direction = Vec3.from_deg_angle(-45)

# transform vector into OCS and get angle of vector in xy-plane

rotation = ucs.to_ocs(text_direction).angle_deg

text = msp.add_text(
    text="TEXT",
    dxfattribs={
        # text rotation angle in degrees in OCS
        'rotation': rotation,
        'extrusion': ucs.uz,
        'thickness': .333,
        'color': 1,
        'style': 'TXT',
    })

# set text position in OCS

text.set_pos(ucs.to_ocs((0, 0, 0)), align='MIDDLE_CENTER')
Hint: For calculating OCS angles from an UCS, be aware that 2D entities, like TEXT or ARC, are placed parallel to the xy-plane of the UCS.

### Placing 2D Arc in 3D Space

Here we have the same problem as for placing text, you need the start and end angle of the arc in degrees in OCS, and this example also shows a shortcut for calculating the OCS angles.

```python
center = msp.add_circle(center=(1, 1, 1), radius=1, dxfattribs={'color': 1})

start = msp.add_line(center=center, end=center + Vec3.from_deg_angle(45), dxfattribs={'color': 1})

end = msp.add_line(center=center, end=center + Vec3.from_deg_angle(270), dxfattribs={'color': 1})
```

```python

# Place arc in 3D space

uc = UCS(origin=(0, 0, 0), ux=(1, 0, 0), uz=(0, 1, 1))
center = msp.add_line(center=uc.to_wcs((0, 0)), end=uc.to_wcs(Vec3.from_deg_angle(45)), dxfattribs={'color': 1})

center = msp.add_line(center=uc.to_wcs((0, 0)), end=uc.to_wcs(Vec3.from_deg_angle(270)), dxfattribs={'color': 1})
```

```python

# Place arc in OCS

uc = UCS(origin=(0, 0, 0), ux=(1, 0, 0), uz=(0, 1, 1))
center = msp.add_line(center=uc.to_wcs((0, 0)), end=uc.to_wcs(Vec3.from_deg_angle(45)), dxfattribs={'color': 1})

center = msp.add_line(center=uc.to_wcs((0, 0)), end=uc.to_wcs(Vec3.from_deg_angle(270)), dxfattribs={'color': 1})
```

```python

# Place arc in OCS

uc = UCS(origin=(0, 0, 0), ux=(1, 0, 0), uz=(0, 1, 1))
center = msp.add_line(center=uc.to_wcs((0, 0)), end=uc.to_wcs(Vec3.from_deg_angle(45)), dxfattribs={'color': 1})

center = msp.add_line(center=uc.to_wcs((0, 0)), end=uc.to_wcs(Vec3.from_deg_angle(270)), dxfattribs={'color': 1})
```

```python

# Place arc in OCS

uc = UCS(origin=(0, 0, 0), ux=(1, 0, 0), uz=(0, 1, 1))
center = msp.add_line(center=uc.to_wcs((0, 0)), end=uc.to_wcs(Vec3.from_deg_angle(45)), dxfattribs={'color': 1})

center = msp.add_line(center=uc.to_wcs((0, 0)), end=uc.to_wcs(Vec3.from_deg_angle(270)), dxfattribs={'color': 1})
```
Placing Block References in 3D Space

Despite the fact that block references (Insert) can contain true 3D entities like Line or Mesh, the Insert entity uses the same placing principle as Text or Arc shown in the previous chapters.

Simple placing by OCS and rotation about the z-axis, can be achieved the same way as for generic 2D entity types. The DXF attribute Insert.dxf.rotation rotates a block reference around the block z-axis, which is located in the Block.dxf.base_point. To rotate the block reference around the WCS x-axis, a transformation of the block z-axis into the WCS x-axis is required by rotating the block z-axis 90 degree counter clockwise around y-axis by using an UCS:

This is just an excerpt of the important parts, see the whole code of insert.py at github.
# rotate UCS around an arbitrary axis:

def ucs_rotation(ucs: UCS, axis: Vec3, angle: float):
    # new in ezdxf v0.11: UCS.rotate(axis, angle)
    t = Matrix44.axis_rotate(axis, math.radians(angle))
    ux, uy, uz = t.transform_vertices([ucs.ux, ucs.uy, ucs.uz])
    return UCS(origin=ucs.origin, ux=ux, uy=uy, uz=uz)

doc = ezdxf.new('R2010', setup=True)
blk = doc.blocks.new('CSYS')
setup_csys(blk)
msp = doc.modelspace()

ucs = ucs_rotation(UCS(), axis=Y_AXIS, angle=90)
# transform insert location to OCS
insert = ucs.to_ocs((0, 0, 0))
# rotation angle about the z-axis (= WCS x-axis)
rotation = ucs.to_ocs_angle_deg(15)
msp.add_blockref('CSYS', insert, dxfattribs={
    'extrusion': ucs.uz,
    'rotation': rotation,
})
To rotate a block reference around another axis than the block z-axis, you have to find the rotated z-axis (extrusion vector) of the rotated block reference, following example rotates the block reference around the block x-axis by 15 degrees:

```python
# t is a transformation matrix to rotate 15 degree around the x-axis
t = Matrix44.axis_rotate(axis=X_AXIS, angle=math.radians(15))
# transform block z-axis into new UCS z-axis (= extrusion vector)
uz = Vec3(t.transform(Z_AXIS))
# create new UCS at the insertion point, because we are rotating around the x-axis,
# ux is the same as the WCS x-axis and uz is the rotated z-axis.
ucs = UCS(origin=(1, 2, 0), ux=X_AXIS, uz=uz)
# transform insert location to OCS, block base_point=(0, 0, 0)
insert = ucs.to_ocs((0, 0, 0))
# for this case a rotation around the z-axis is not required
rotation = 0
blockref = msp.add_blockref('CSYS', insert, dxfattribs={
    'extrusion': ucs.uz,
    'rotation': rotation,
})
```
The next example shows how to translate a block references with an already established OCS:

```python
# translate a block references with an established OCS
translation = Vec3(-3, -1, 1)
# get established OCS
ocs = blockref.ocs()
# get insert location in WCS
actual_wcs_location = ocs.to_wcs(blockref.dxf.insert)
# translate location
new_wcs_location = actual_wcs_location + translation
# convert WCS location to OCS location
blockref.dxf.insert = ocs.from_wcs(new_wcs_location)
```

Setting a new insert location is the same procedure without adding a translation vector, just transform the new insert location into the OCS.
The next operation is to rotate a block reference with an established OCS, rotation axis is the block y-axis, rotation angle is -90 degrees. First transform block y-axis (rotation axis) and block z-axis (extrusion vector) from OCS into WCS:

```python
# rotate a block references with an established OCS around the block y-axis about -90 degree
ocs = blockref.ocs()
# convert block y-axis (= rotation axis) into WCS vector
rotation_axis = ocs.to_wcs((0, 1, 0))
# convert local z-axis (=extrusion vector) into WCS vector
local_z_axis = ocs.to_wcs((0, 0, 1))
```

Build transformation matrix and transform extrusion vector and build new UCS:

```python
# build transformation matrix
```
t = Matrix44.axis_rotate(axis=rotation_axis, angle=math.radians(-90))
uz = t.transform(local_z_axis)
uy = rotation_axis
# the block reference origin stays at the same location, no rotation needed
wcs_insert = ocs.to_wcs(blockref.dxf.insert)
# build new UCS to convert WCS locations and angles into OCS
ucs = UCS(origin=wcs_insert, uy=uy, uz=uz)

Set new OCS attributes, we also have to set the rotation attribute even though we do not rotate the block reference around the local z-axis, the new block x-axis (0 deg) differs from OCS x-axis and has to be adjusted:

# set new OCS
blockref.dxf.extrusion = ucs.uz
# set new insert
blockref.dxf.insert = ucs.to_ocs((0, 0, 0))
# set new rotation: we do not rotate the block reference around the local z-axis,
# but the new block x-axis (0 deg) differs from OCS x-axis and has to be adjusted
blockref.dxf.rotation = ucs.to_ocs_angle_deg(0)
And here is the point, where my math knowledge ends, for more advanced CAD operation you have to look elsewhere.

### 6.5.18 Tutorial for UCS Based Transformations

With ezdxf v0.11 a new feature for entity transformation was introduced, which makes working with OCS/UCS much easier, this is a new edition of the older Tutorial for OCS/UCS Usage. For the basic information read the old tutorial please. In ezdxf v0.13 the transform_to_wcs() interface was replaced by the general transformation interface: transform().

For this tutorial we don’t have to worry about the OCS and the extrusion vector, this is done automatically by the transform() method of each DXF entity.

**Placing 2D Circle in 3D Space**

To recreate the situation of the old tutorial instantiate a new UCS and rotate it around the local x-axis. Use UCS coordinates to place the 2D CIRCLE in 3D space, and transform the UCS coordinates to the WCS.

```python
import math
import ezdxf
from ezdxf.math import UCS

doc = ezdxf.new('R2010')
msp = doc.modelspace()

ucs = UCS()  # New default UCS
# All rotation angles in radians, and rotation
# methods always return a new UCS.
ucs = ucs.rotate_local_x(math.radians(-45))
circle = msp.add_circle(
    # Use UCS coordinates to place the 2d circle in 3d space
    center=(0, 0, 2),
    radius=1,
    dxfattribs={'color': 1}
)
```

(continues on next page)
circle.transform(ucs.matrix)

# mark center point of circle in WCS
msp.add_point((0, 0, 2), dxfattribs={'color': 1}).transform(ucs.matrix)

---

**Placing LWPolyline in 3D Space**

Simplified LWPOLYLINE example:

```python
# The center of the pentagon should be (0, 2, 2), and the shape is rotated around x-axis about -45 degree
ucx = UCS(origin=(0, 2, 2)).rotate_local_x(math.radians(-45))
```
msp.add_lwpolyline(
    # calculating corner points in UCS coordinates
    points=(Vec3.from_deg_angle((360 / 5) * n) for n in range(5)),
    format='xy',  # ignore z-axis
    close=True,
    dxfattribs={
        'color': 1,
    }
)

The 2D pentagon in 3D space in BricsCAD Left and Front view.
Using UCS to Place 3D Polyline

Simplified POLYLINE example: Using a first UCS to transform the POLYLINE and a second UCS to place the POLYLINE in 3D space.

```python
# using an UCS simplifies 3D operations, but UCS definition can happen later
# calculating corner points in local (UCS) coordinates without Vec3 class
angle = math.radians(360 / 5)
corners_ucs = [(math.cos(angle * n), math.sin(angle * n), 0) for n in range(5)]

# let's do some transformations by UCS
transformation_ucs = UCS().rotate_local_z(math.radians(15))  # 1. rotation around z-axis
transformation_ucs.shift((0, .333, .333))  # 2. translation (inplace)
corners_ucs = list(transformation_ucs.points_to_wcs(corners_ucs))

location_ucs = UCS(origin=(0, 2, 2)).rotate_local_x(math.radians(-45))
msp.add_polyline3d(
    points=corners_ucs,
    close=True,
    dxfattribs={
        'color': 1,
    }
).transform(location_ucs.matrix)

# Add lines from the center of the POLYLINE to the corners
center_ucs = transformation_ucs.to_wcs((0, 0, 0))
for corner in corners_ucs:
    msp.add_line(
        center_ucs, corner, dxfattribs={'color': 1}
    ).transform(location_ucs.matrix)
```
Placing 2D Text in 3D Space

The problem with the text rotation in the old tutorial disappears (or better it is hidden in `transform()`) with the new UCS based transformation method:

AutoCAD supports thickness for the TEXT entity only for .shx fonts and not for true type fonts.

```python
# thickness for text works only with shx fonts not with true type fonts
doc.styles.new('TXT', dxfattribs={'font': 'romans.shx'})

ucs = UCS(origin=(0, 2, 2)).rotate_local_x(math.radians(-45))
text = msp.add_text(text="TEXT",
                      dxfattribs={
                          # text rotation angle in degrees in UCS
                          'rotation': -45,
                          'thickness': .333,
                          'color': 1,
                          'style': 'TXT',
                      })

# set text position in UCS
text.set_pos((0, 0, 0), align='MIDDLE_CENTER')
text.transform(ucs.matrix)
```
Placing 2D Arc in 3D Space

Same as for the text example, OCS angle transformation can be ignored:

```python
ucs = UCS(origin=(0, 2, 2)).rotate_local_x(math.radians(-45))
CENTER = (0, 0)
START_ANGLE = 45
END_ANGLE = 270
msp.add_arc(
    center=CENTER,
    radius=1,
    start_angle=START_ANGLE,
    end_angle=END_ANGLE,
    dxfattribs={'color': 6},
).transform(ucs.matrix)
```
msp.add_line(
    start=Center,
    end=Vec3.from_deg_angle(START_ANGLE),
    dxfattribs={'color': 6},
).transform(ucs.matrix)

msp.add_line(
    start=Center,
    end=Vec3.from_deg_angle(END_ANGLE),
    dxfattribs={'color': 6},
).transform(ucs.matrix)
Placing Block References in 3D Space

Despite the fact that block references (INSERT) can contain true 3D entities like LINE or MESH, the INSERT entity uses the same placing principle as TEXT or ARC shown in the previous chapters.

To rotate the block reference 15 degrees around the WCS x-axis, we place the block reference in the origin of the UCS, and rotate the UCS 90 degrees around its local y-axis, to align the UCS z-axis with the WCS x-axis:

This is just an excerpt of the important parts, see the whole code of insert.py at github.

```python
doc = ezdxf.new('R2010', setup=True)
blk = doc.blocks.new('CSYS')
setup_csys(blk)
msp = doc.modelspace()

ucs = UCS().rotate_local_y(angle=math.radians(90))
msp.add_blockref(
    'CSYS',
    insert=(0, 0),
    # rotation around the block z-axis (= WCS x-axis)
    dxfattribs=('rotation': 15),
).transform(ucs.matrix)
```

![Diagram of 3D placement and rotation of block references.]
A more simple approach is to ignore the `rotate` attribute at all and just rotate the UCS. To rotate a block reference around any axis rather than the block z-axis, rotate the UCS into the desired position. Following example rotates the block reference around the block x-axis by 15 degrees:

```python
ucs = UCS(origin=(1, 2, 0)).rotate_local_x(math.radians(15))
blockref = msp.add_blockref('CSYS', insert=(0, 0, 0))
blockref.transform(ucs.matrix)
```
The next example shows how to translate a block references with an already established OCS:

```python
# New UCS at the translated location, axis aligned to the WCS
ucs = UCS((-3, -1, 1))
# Transform an already placed block reference, including
# the transformation of the established OCS.
blockref.transform(ucs.matrix)
```
The next operation is to rotate a block reference with an established OCS, rotation axis is the block y-axis, rotation angle is -90 degrees. The idea is to create an UCS in the origin of the already placed block reference, UCS axis aligned to the block axis and resetting the block reference parameters for a new WCS transformation.

```python
# Get UCS at the block reference insert location, UCS axis aligned
# to the block axis.
ucs = blockref.ucs()
# Rotate UCS around the local y-axis.
ucs = ucs.rotate_local_y(math.radians(-90))
```

Reset block reference parameters, this places the block reference in the UCS origin and aligns the block axis to the UCS axis, now we do a new transformation from UCS to WCS:

```python
# Reset block reference parameters to place block reference in
# UCS origin, without any rotation and OCS.
blockref.reset_transformation()

# Transform block reference from UCS to WCS
blockref.transform(ucs.matrix)
```
6.5.19 Tutorial for Linear Dimensions

The Dimension entity is the generic entity for all dimension types, but unfortunately AutoCAD is not willing to show a dimension line defined only by this dimension entity, it also needs an anonymous block which contains the dimension line shape constructed by basic DXF entities like LINE and TEXT entities, this representation is called the dimension line rendering in this documentation, beside the fact this is not a real graphical rendering. BricsCAD is a much more friendly CAD application, which do show the dimension entity without the graphical rendering as block, which was very useful for testing, because there is no documentation how to apply all the dimension style variables (more than 80). This seems to be the reason why dimension lines are rendered so differently by many CAD application.

Don’t expect to get the same rendering results by ezdxf as you get from AutoCAD, ezdxf tries to be as close to the results rendered by BricsCAD, but it was not possible to implement all the various combinations of dimension style parameters.
Text rendering is another problem, because ezdxf has no real rendering engine. Some font properties, like the real text width, are not available to ezdxf and may also vary slightly for different CAD applications. The text properties in ezdxf are based on the default monospaced standard font, but for TrueType fonts the space around the text is much bigger than needed.

Not all DIMENSION and DIMSTYLE features are supported by all DXF versions, especially DXF R12 does not support many features, but in this case the required rendering of dimension lines is an advantage, because if the application just shows the rendered block, all features which can be used in DXF R12 are displayed like linetypes, but they disappear if the dimension line is edited in the application. ezdxf writes only the supported DIMVARS of the used DXF version to avoid invalid DXF files. So it is not that critical to know all the supported features of a DXF version, except for limits and tolerances, ezdxf uses the advanced features of MTEXT to create limits and tolerances and therefore they are not supported (displayed) in DXF R12 files.

See also:

- Graphical reference of many DIMVARS and some advanced information: DIMSTYLE Table
- Source code file standards.py shows how to create your own DIMSTYLES.
- dimension_linear.py for linear dimension examples.

**Horizontal Dimension**

```python
import ezdxf

# Argument setup=True setups the default dimension styles
doc = ezdxf.new('R2010', setup=True)

# Add new dimension entities to the modelspace
msp = doc.modelspace()

# Add a LINE entity, not required
msp.add_line((0, 0), (3, 0))

# Add a horizontal dimension, default dimension style is 'EZDXF'
dim = msp.add_linear_dim(base=(3, 2), p1=(0, 0), p2=(3, 0))

# Necessary second step, to create the BLOCK entity with the dimension geometry.
# Additional processing of the dimension line could happen between adding and
# rendering call.
dim.render()
doc.saveas('dim_linear_horiz.dxf')
```
The example above creates a horizontal `Dimension` entity, the default dimension style 'EZDXF' and is defined as 1 drawing unit is 1 m in reality, the drawing scale 1:100 and the length factor is 100, which creates a measurement text in cm.

The `base` point defines the location of the dimension line, `ezdxf` accepts any point on the dimension line, the point `p1` defines the start point of the first extension line, which also defines the first measurement point and the point `p2` defines the start point of the second extension line, which also defines the second measurement point.

The return value `dim` is not a dimension entity, instead a `DimStyleOverride` object is returned, the dimension entity is stored as `dim.dimension`.

**Vertical and Rotated Dimension**

Argument `angle` defines the angle of the dimension line in relation to the x-axis of the WCS or UCS, measurement is the distance between first and second measurement point in direction of `angle`.

```python
# assignment to dim is not necessary, if no additional processing happens
msp.add_linear_dim(base=(3, 2), p1=(0, 0), p2=(3, 0), angle=-30).render()
doc.saveas('dim_linear_rotated.dxf')
```
For a vertical dimension set argument *angle* to 90 degree, but in this example the vertical distance would be 0.

**Aligned Dimension**

An aligned dimension line is parallel to the line defined by the definition points *p1* and *p2*. The placement of the dimension line is defined by the argument *distance*, which is the distance between the definition line and the dimension line. The *distance* of the dimension line is orthogonal to the base line in counter clockwise orientation.

```python
msp.add_line((0, 2), (3, 0))
dim = msp.add_aligned_dim(p1=(0, 2), p2=(3, 0), distance=1)
doc.saveas('dim_linear_aligned.dxf')```
**Dimension Style Override**

Many dimension styling options are defined by the associated `DimStyle` entity. But often you wanna change just a few settings without creating a new dimension style, therefore the DXF format has a protocol to store this changed settings in the dimension entity itself. This protocol is supported by `ezdxf` and every factory function which creates dimension entities supports the `override` argument. This `override` argument is a simple Python dictionary (e.g. `override = {'dimtad': 4}`, place measurement text below dimension line).

The overriding protocol is managed by the `DimStyleOverride` object, which is returned by the most dimension factory functions.

**Placing Measurement Text**

The “default” location of the measurement text depends on various `DimStyle` parameters and is applied if no user defined text location is defined.

**Default Text Locations**

“Horizontal direction” means in direction of the dimension line and “vertical direction” means perpendicular to the dimension line direction.
The “**horizontal**” location of the measurement text is defined by *dimjust*:

<table>
<thead>
<tr>
<th>dimjust</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Center of dimension line</td>
</tr>
<tr>
<td>1</td>
<td>Left side of the dimension line, near first extension line</td>
</tr>
<tr>
<td>2</td>
<td>Right side of the dimension line, near second extension line</td>
</tr>
<tr>
<td>3</td>
<td>Over first extension line</td>
</tr>
<tr>
<td>4</td>
<td>Over second extension line</td>
</tr>
</tbody>
</table>

```python
dsp.add_linear_dim(base=(3, 2), p1=(0, 0), p2=(3, 0), override={'dimjust': 1}).render()
```

The “**vertical**” location of the measurement text relative to the dimension line is defined by *dimtad*:

<table>
<thead>
<tr>
<th>dimtad</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Center, it is possible to adjust the vertical location by <em>dimtvp</em></td>
</tr>
<tr>
<td>1</td>
<td>Above</td>
</tr>
<tr>
<td>2</td>
<td>Outside, handled like <em>Above</em> by ezdxf</td>
</tr>
<tr>
<td>3</td>
<td>JIS, handled like <em>Above</em> by ezdxf</td>
</tr>
<tr>
<td>4</td>
<td>Below</td>
</tr>
</tbody>
</table>

```python
dsp.add_linear_dim(base=(3, 2), p1=(0, 0), p2=(3, 0), override={'dimtad': 4}).render()
```

The distance between text and dimension line is defined by *dimgap*.

The *DimStyleOverride* object has a method *set_text_align()* to set the default text location in an easy way, this is also the reason for the 2 step creation process of dimension entities:

```python
dim = dsp.add_linear_dim(base=(3, 2), p1=(0, 0), p2=(3, 0))
dim.set_text_align(halign='left', valign='center')
dim.render()
```

```
halign = ['left', 'right', 'center', 'above1', 'above2']
valign = ['above', 'center', 'below']
```
Run function `example_for_all_text_placings_R2007()` in the example script `dimension_linear.py` to create a DXF file with all text placings supported by `ezdxf`.

**User Defined Text Locations**

Beside the default location, it is possible to locate the measurement text freely.

**Location Relative to Origin**

The user defined text location can be set by the argument `location` in most dimension factory functions and always references the midpoint of the measurement text:

```
msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0), location=(4, 4)).render()
```

The `location` is relative to origin of the active coordinate system or WCS if no UCS is defined in the `render()` method, the user defined `location` can also be set by `user_location_override()`.

**Location Relative to Center of Dimension Line**

The method `set_location()` has additional features for linear dimensions. Argument `leader = True` adds a simple leader from the measurement text to the center of the dimension line and argument `relative = True` places the
measurement text relative to the center of the dimension line.

```python
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_location(location=(-1, 1), leader=True, relative=True)
dim.render()
```

**Location Relative to Default Location**

The method `shift_text()` shifts the measurement text away from the default text location. Shifting directions are aligned to the text direction, which is the direction of the dimension line in most cases. `dh` (for delta horizontal) shifts the text parallel to the text direction, `dv` (for delta vertical) shifts the text perpendicular to the text direction. This method does not support leaders.

```python
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.shift_text(dh=1, dv=1)
dim.render()
```
**Measurement Text Formatting and Styling**

**Text Properties**

<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dimtxsty</td>
<td>Specifies the text style of the dimension as <em>Textstyle</em> name.</td>
</tr>
<tr>
<td>dimtxt</td>
<td>Text height in drawing units.</td>
</tr>
<tr>
<td>dimclrt</td>
<td>Measurement text color as <em>AutoCAD Color Index (ACI).</em></td>
</tr>
</tbody>
</table>

```python
msp.add_linear_dim(
    base=(3, 2), p1=(3, 0), p2=(6, 0),
    override={
        'dimtxsty': 'Standard',
        'dimtxt': 0.35,
        'dimclrt': 1,
    }).render()
```
Background Filling

Background fillings are supported since DXF R2007, and ezdxf uses the MTEXT entity to implement this feature, so setting background filling in DXF R12 has no effect.

Set `dimtfill` to 1 to use the canvas color as background filling or set `dimtfill` to 2 to use `dimtfillclr` as background filling, color value as *AutoCAD Color Index (ACI)*. Set `dimtfill` to 0 to disable background filling.

<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dimtfill</td>
<td>Enables background filling if bigger than 0</td>
</tr>
<tr>
<td>dimtfillclr</td>
<td>Fill color as <em>AutoCAD Color Index (ACI)</em>, if <code>dimtfill</code> is 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dimtfill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>disabled</td>
</tr>
<tr>
<td>1</td>
<td>canvas color</td>
</tr>
<tr>
<td>2</td>
<td>color defined by <code>dimtfillclr</code></td>
</tr>
</tbody>
</table>

```python
msp.add_linear_dim(
    base=(3, 2), p1=(3, 0), p2=(6, 0),
    override={
        'dimtfill': 2,
        'dimtfillclr': 1,
    }).render()
```

Text Formatting

- Set decimal places: `dimdec` defines the number of decimal places displayed for the primary units of a dimension. (DXF R2000)
- Set decimal point character: `dimdsep` defines the decimal point as ASCII code, use `ord('.')`
- Set rounding: `dimrnd`, rounds all dimensioning distances to the specified value, for instance, if `dimrnd` is set to 0.25, all distances round to the nearest 0.25 unit. If `dimrnd` is set to 1.0, all distances round to the nearest integer. For more information look at the documentation of the `ezdxf.math.xround()` function.
- Set zero trimming: `dimzin`, ezdxf supports only: 4 suppress leading zeros and 8: suppress trailing zeros and both as 12.
- Set measurement factor: scale measurement by factor `dimlfac`, e.g. to get the dimensioning text in cm for a DXF file where 1 drawing unit represents 1m, set `dimlfac` to 100.
- Text template for measurement text is defined by `dimpost`, `<>` represents the measurement text, e.g. `~<>cm` produces `~300cm` for measurement in previous example.
To set this values the `ezdxf.entities.DimStyle.set_text_format()` and `ezdxf.entities.DimStyleOverride.set_text_format()` methods are very recommended.

**Overriding Measurement Text**

Measurement text overriding is stored in the `Dimension` entity, the content of to DXF attribute `text` represents the override value as string. Special values are one space `' '` to just suppress the measurement text, an empty string `''` or `'<>'` to get the regular measurement.

All factory functions have an explicit `text` argument, which always replaces the `text` value in the `dxfattribs` dict.

```python
msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0), text='>1m').render()
```

---

**Dimension Line Properties**

The dimension line color is defined by the DIMVAR `dimclrd` as *AutoCAD Color Index (ACI)*, `dimclrd` also defines the color of the arrows. The linetype is defined by `dimltype` but requires DXF R2007 for full support by CAD Applications and the line weight is defined by `dimlwd` (DXF R2000), see also the `lineweight` reference for valid values. The `dimdle` is the extension of the dimension line beyond the extension lines, this dimension line extension is not supported for all arrows.

<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dimclrd</td>
<td>dimension line and arrows color as <em>AutoCAD Color Index (ACI)</em></td>
</tr>
<tr>
<td>dimltype</td>
<td>linetype of dimension line</td>
</tr>
<tr>
<td>dimlwd</td>
<td>line weight of dimension line</td>
</tr>
<tr>
<td>dimdle</td>
<td>extension of dimension line in drawing units</td>
</tr>
</tbody>
</table>

```python
msp.add_linear_dim(
    base=(3, 2), p1=(3, 0), p2=(6, 0),
    override={
        'dimclrd': 1,  # red
        'dimdle': 0.25,
        'dimltype': 'DASHED2',
        'dimlwd': 35,  # 0.35mm line weight
    }).render()
```
**DimStyleOverride() method:**

```python
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_dimline_format(color=1, linetype='DASHED2', lineweight=35, extension=0.25)
dim.render()
```

**Extension Line Properties**

The extension line color is defined by the DIMVAR `dimclre` as *AutoCAD Color Index (ACI)*. The linetype for first and second extension line is defined by `dimltx1` and `dimltx2` but requires DXF R2007 for full support by CAD Applications and the line weight is defined by `dimlwe` (DXF R2000), see also the `lineweight` reference for valid values.

The `dimexe` is the extension of the extension line beyond the dimension line, and `dimexo` defines the offset of the extension line from the measurement point.

<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dimclre</code></td>
<td>extension line color as <em>AutoCAD Color Index (ACI)</em></td>
</tr>
<tr>
<td><code>dimltx1</code></td>
<td>linetype of first extension line</td>
</tr>
<tr>
<td><code>dimltx2</code></td>
<td>linetype of second extension line</td>
</tr>
<tr>
<td><code>dimlwe</code></td>
<td>line weight of extension line</td>
</tr>
<tr>
<td><code>dimexe</code></td>
<td>extension beyond dimension line in drawing units</td>
</tr>
<tr>
<td><code>dimexo</code></td>
<td>offset of extension line from measurement point</td>
</tr>
<tr>
<td><code>dimfxlon</code></td>
<td>set to 1 to enable fixed length extension line</td>
</tr>
<tr>
<td><code>dimfxl</code></td>
<td>length of fixed length extension line in drawing units</td>
</tr>
<tr>
<td><code>dimsel1</code></td>
<td>suppress first extension line if 1</td>
</tr>
<tr>
<td><code>dimse2</code></td>
<td>suppress second extension line if 1</td>
</tr>
</tbody>
</table>

```python
msp.add_linear_dim(
    base=(3, 2), p1=(3, 0), p2=(6, 0),
    override={
        'dimclre': 1, # red
        'dimltx1': 'DASHED2',
        'dimltx2': 'CENTER2',
        'dimlwe': 35, # 0.35mm line weight
        'dimexe': 0.3, # length above dimension line
        'dimexo': 0.1, # offset from measurement point
    }).render()
```
DimStyleOverride() methods:

```python
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_extline_format(color=1, lineweight=35, extension=0.3, offset=0.1)
dim.set_extline1(linetype='DASHED2')
dim.set_extline2(linetype='CENTER2')
dim.render()
```

Fixed length extension lines are supported in DXF R2007+, set dimfxlon to 1 and dimfxl defines the length of the extension line starting at the dimension line.

```python
msp.add_linear_dim(
    base=(3, 2), p1=(3, 0), p2=(6, 0),
    override={
        'dimfxlon': 1,  # fixed length extension lines
        'dimexe': 0.2,  # length above dimension line
        'dimfxl': 0.4,  # length below dimension line
    }).render()
```
DimStyleOverride() method:

```python
dim = msp.add_linear_dim(base=(3, 2), pl=(3, 0), p2=(6, 0))
dim.set_extline_format(extension=0.2, fixed_length=0.4)
dim.render()
```

To suppress extension lines set `dimse1 = 1` to suppress the first extension line and `dimse2 = 1` to suppress the second extension line.

```python
msp.add_linear_dim(
    base=(3, 2), pl=(3, 0), p2=(6, 0),
    override={
        'dimse1': 1,  # suppress first extension line
        'dimse2': 1,  # suppress second extension line
        'dimblk': ezdxf.ARROWS.closed_filled,  # arrows just looks better
    }).render()
```

DimStyleOverride() methods:

```
dim = msp.add_linear_dim(base=(3, 2), pl=(3, 0), p2=(6, 0))
dim.set_arrows(blk=ezdxf.ARROWS.closed_filled)
dim.set_extline1(disable=True)
dim.set_extline2(disable=True)
dim.render()
```

Arrows

“Arrows” mark then beginning and the end of a dimension line, and most of them do not look like arrows.
DXF distinguish between the simple tick and arrows as blocks.

Using the simple tick by setting tick size `dimtsz` != 0 also disables arrow blocks as side effect:

```python
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_tick(size=0.25)
dim.render()
```

`ezdxf` uses the "ARCHTICK" block at double size to render the tick (AutoCAD and BricsCad just draw a simple line), so there is no advantage of using the tick instead of an arrow.

Using arrows:

```python
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_arrow(blk="OPEN_30", size=0.25)
dim.render()
```

### DIMVAR

<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dimtsz</td>
<td>tick size in drawing units, set to 0 to use arrows</td>
</tr>
<tr>
<td>dimblk</td>
<td>set both arrow block names at once</td>
</tr>
<tr>
<td>dimblk1</td>
<td>first arrow block name</td>
</tr>
<tr>
<td>dimblk2</td>
<td>second arrow block name</td>
</tr>
<tr>
<td>dimasz</td>
<td>arrow size in drawing units</td>
</tr>
</tbody>
</table>

Dimension line extension (dimdle) works only for a few arrow blocks and the simple tick:

- "ARCHTICK"
- "OBLIQUE"
- "NONE"
- "SMALL"
- "DOTSMALL"
- "INTEGRAL"
Arrow Shapes

The arrow names are stored as attributes in the ezdxf.ARROWS object.

Arrow Names

The arrow names are stored as attributes in the ezdxf.ARROWS object.
Tolerances and Limits

The tolerances ans limits features are implemented by using the \texttt{MText} entity, therefore DXF R2000+ is required to use these features. It is not possible to use both tolerances and limits at the same time.

Tolerances

Geometrical tolerances are shown as additional text appended to the measurement text. It is recommend to use \texttt{set_tolerance()} method in \texttt{DimStyleOverride} or \texttt{DimStyle}.

The attribute \texttt{dimtp} defines the upper tolerance value, \texttt{dimtm} defines the lower tolerance value if present, else the lower tolerance value is the same as the upper tolerance value. Tolerance values are shown as given!

Same upper and lower tolerance value:

\begin{verbatim}
dim = msp.add_linear_dim(base=(0, 3), p1=(3, 0), p2=(6.5, 0))
dim.set_tolerance(.1, hfactor=.4, align="top", dec=2)
dim.render()
\end{verbatim}
Different upper and lower tolerance values:

```python
dim = msp.add_linear_dim(base=(0, 3), pl=(3, 0), p2=(6.5, 0))
dim.set_tolerance(upper=.1, lower=.15, hfactor=.4, align="middle", dec=2)
dim.render()
```

The attribute `dimtfac` specifies a scale factor for the text height of limits and tolerance values relative to the dimension text height, as set by `dimtxt`. For example, if `dimtfac` is set to 1.0, the text height of fractions and tolerances is the same height as the dimension text. If `dimtxt` is set to 0.75, the text height of limits and tolerances is three-quarters the size of dimension text.

Vertical justification for tolerances is specified by `dimtolj`:

<table>
<thead>
<tr>
<th>dimtolj</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Align with bottom line of dimension text</td>
</tr>
<tr>
<td>1</td>
<td>Align vertical centered to dimension text</td>
</tr>
<tr>
<td>2</td>
<td>Align with top line of dimension text</td>
</tr>
<tr>
<td>DIMVAR</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>dimtol</td>
<td>set to 1 to enable tolerances</td>
</tr>
<tr>
<td>dimtp</td>
<td>set the maximum (or upper) tolerance limit for dimension text</td>
</tr>
<tr>
<td>dimtm</td>
<td>set the minimum (or lower) tolerance limit for dimension text</td>
</tr>
<tr>
<td>dimtfac</td>
<td>specifies a scale factor for the text height of limits and tolerance values relative to the dimension text height, as set by dimtxt.</td>
</tr>
<tr>
<td>dimtzin</td>
<td>set the number of decimal places to display in tolerance values</td>
</tr>
<tr>
<td>dimtolj</td>
<td>set the vertical justification for tolerance values relative to the nominal dimension text.</td>
</tr>
<tr>
<td>dimtdec</td>
<td>set the vertical justification for tolerance values relative to the nominal dimension text.</td>
</tr>
</tbody>
</table>

## Limits

The geometrical limits are shown as upper and lower measurement limit and replaces the usual measurement text. It is recommend to use `set_limits()` method in `DimStyleOverride` or `DimStyle`.

For limits the tolerance values are drawing units scaled by measurement factor `dimlfac`, the upper limit is scaled measurement value + `dimtp` and the lower limit is scaled measurement value - `dimtm`.

The attributes `dimtfac`, `dimtzin` and `dimtdec` have the same meaning for limits as for tolerances.

```python
dim = msp.add_linear_dim(base=(0, 3), p1=(3, 0), p2=(6.5, 0))
dim.set_limits(upper=.1, lower=.15, hfactor=.4, dec=2)
dim.render()
```

## Alternative Units

Alternative units are not supported.

### 6.5.20 Tutorial for Radius Dimensions

Please read the *Tutorial for Linear Dimensions* before, if you haven’t.
import ezdxf

# DXF R2010 drawing, official DXF version name: 'AC1024',
# setup=True setups the default dimension styles
doc = ezdxf.new('R2010', setup=True)

msp = doc.modelspace()  # add new dimension entities to the modelspace
msp.add_circle((0, 0), radius=3)  # add a CIRCLE entity, not required
# add default radius dimension, measurement text is located outside
dim = msp.add_radius_dim(center=(0, 0), radius=3, angle=45, dimstyle='EZ_RADIUS')
# necessary second step, to create the BLOCK entity with the dimension geometry.
dim.render()
doc.saveas('radius_dimension.dxf')

The example above creates a 45 degrees slanted radius Dimension entity, the default dimension style 'EZ_RADIUS' is defined as 1 drawing unit is 1m in reality, drawing scale 1:100 and the length factor is 100, which creates a measurement text in cm, the default location for the measurement text is outside of the circle.

The center point defines the the center of the circle but there doesn’t have to exist a circle entity, radius defines the circle radius, which is also the measurement, and angle defines the slope of the dimension line, it is also possible to define the circle by a measurement point mpoint on the circle.

The return value \textit{dim} is not a dimension entity, instead a \texttt{DimStyleOverride} object is returned, the dimension entity is stored as \texttt{dim.dimension}.

### Placing Measurement Text

There are different predefined DIMSTYLES to achieve various text placing locations.

DIMSTYLE 'EZ_RADIUS' settings are: 1 drawing unit is 1m, scale 1:100, length_factor is 100 which creates measurement text in cm, and a closed filled arrow with size 0.25 is used.

**Note:** Not all possibles features of DIMSTYLE are supported and especially for radial dimension there are less features supported as for linear dimension because of the lack of good documentation.

**See also:**
- Graphical reference of many DIMVARS and some advanced information: \textit{DIMSTYLE Table}
- Source code file \texttt{standards.py} shows how to create your own DIMSTYLES.
- \texttt{dimension_radius.py} for radius dimension examples.

### Default Text Locations Outside

'EZ_RADIUS' default settings for to place text outside:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tmove</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>dim-tad</strong></td>
<td>1</td>
</tr>
<tr>
<td>to keep dim line with text, this is the best setting for text outside to preserve appearance of the DIMENSION entity, if editing afterwards in BricsCAD or AutoCAD.</td>
<td></td>
</tr>
<tr>
<td>to place text vertical above the dimension line</td>
<td></td>
</tr>
</tbody>
</table>
dim = msp.add_radius_dim(center=(0, 0), radius=2.5, angle=45, 
    dimstyle='EZ_RADIUS')

dim.render()  # required, but not shown in the following examples

To force text outside horizontal set \texttt{dimtoh} to 1:

dim = msp.add_radius_dim(center=(0, 0), radius=2.5, angle=45, 
    dimstyle='EZ_RADIUS', 
    override={'dimtoh': 1})

\textbf{Default Text Locations Inside}

\texttt{DIMSTYLE 'EZ_RADIUS_INSIDE'} can be used to place the dimension text inside the circle at a default location. Default \texttt{DIMSTYLE} settings are: 1 drawing unit is 1m, scale 1:100, length factor is 100 which creates measurement text in cm, and a closed filled arrow with size 0.25 is used.

\texttt{'EZ_RADIUS_INSIDE'} default settings:
tmove: 0 to keep dim line with text, this is the best setting for text inside to preserve appearance of the DIMENSION entity, if editing afterwards in BricsCAD or AutoCAD.

dimtix: 1 to force text inside

dimmat-fit: 0 to force text inside, required by BricsCAD and AutoCAD

dimtad: 0 to center text vertical, BricsCAD and AutoCAD always create vertical centered text, ezdxf let you choose the vertical placement (above, below, center), but editing the DIMENSION in BricsCAD or AutoCAD will reset text to center placement.

dim = msp.add_radius_dim(center=(0, 0), radius=2.5, angle=45,
                          dimstyle='EZ_RADIUS_INSIDE')

To force text inside horizontal set dimtih to 1:

dim = msp.add_radius_dim(center=(0, 0), radius=2.5, angle=45,
                          dimstyle='EZ_RADIUS_INSIDE',
                          override={'dimtih': 1})
User Defined Text Locations

Beside the default location it is always possible to override the text location by a user defined location. This location also determines the angle of the dimension line and overrides the argument *angle*. For user defined locations it is not necessary to force text inside (*dimtix=1*), because the location of the text is explicit given, therefore the DIMSTYLE 'EZ_RADIUS' can be used for all this examples.

User defined location outside of the circle:

```python
dim = msp.add_radius_dim(center=(0, 0), radius=2.5, location=(4, 4),
                           dimstyle='EZ_RADIUS',
                      override={'dimtoh': 1})
```

User defined location outside of the circle and forced horizontal text:

```python
dim = msp.add_radius_dim(center=(0, 0), radius=2.5, location=(4, 4),
                           dimstyle='EZ_RADIUS',
                      override={'dimtoh': 1})
```
User defined location inside of the circle:

```python
dim = msp.add_radius_dim(center=(0, 0), radius=2.5, location=(1, 1),
                        dimstyle='EZ_RADIUS')
```

User defined location inside of the circle and forced horizontal text:

```python
dim = msp.add_radius_dim(center=(0, 0), radius=2.5, location=(1, 1),
                        dimstyle='EZ_RADIUS',
                        dimtad=1, dimtmove=0)
```

```python
dim = msp.add_radius_dim(center=(0, 0), radius=2.5, location=(1, 1),
                        dimstyle='EZ_RADIUS',
                        dimtad=0, dimtmove=0)
```

```python
dim = msp.add_radius_dim(center=(0, 0), radius=2.5, location=(1, 1),
                        dimstyle='EZ_RADIUS',
                        dimtad=4, dimtmove=0)
```

User defined location inside of the circle and forced horizontal text:
Center Mark/Lines

Center mark/lines are controlled by \texttt{dimcen}, default value is 0 for predefined dimstyles 'EZ_RADIUS' and 'EZ_RADIUS_INSIDE':

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Center mark is off</td>
</tr>
<tr>
<td>&gt;0</td>
<td>Create center mark of given size</td>
</tr>
<tr>
<td>&lt;0</td>
<td>Create center lines</td>
</tr>
</tbody>
</table>

```python
dim = msp.add_radius_dim(center=(0, 0), radius=2.5, angle=45,  
dimstyle='EZ_RADIUS',  
override={'dimcen': 0.25},  
)```

6.5. Tutorials
Overriding Measurement Text

See Linear Dimension Tutorial: *Overriding Measurement Text*

Measurement Text Formatting and Styling

See Linear Dimension Tutorial: *Measurement Text Formatting and Styling*

6.5.21 Tutorial for Diameter Dimensions

Please read the *Tutorial for Radius Dimensions* before, if you haven’t.

This is a repetition of the radius tutorial, just with diameter dimensions.

```python
import ezdxf

# setup=True setups the default dimension styles
doc = ezdxf.new('R2010', setup=True)

msp = doc.modelspace()  # add new dimension entities to the modelspace
msp.add_circle((0, 0), radius=3)  # add a CIRCLE entity, not required
# add default diameter dimension, measurement text is located outside
dim = msp.add_diameter_dim(center=(0, 0), radius=3, angle=45, dimstyle='EZ_RADIUS')
dim.render()
doc.saveas('diameter_dimension.dxf')
```

The example above creates a 45 degrees slanted diameter *Dimension* entity, the default dimension style 'EZ_RADIUS' (same as for radius dimensions) is defined as 1 drawing unit is 1m in reality, drawing scale 1:100 and the length factor is 100, which creates a measurement text in cm, the default location for the measurement text is outside of the circle.

The *center* point defines the the center of the circle but there doesn’t have to exist a circle entity, *radius* defines the circle radius and *angle* defines the slope of the dimension line, it is also possible to define the circle by a measurement point *mpoint* on the circle.

The return value *dim* is not a dimension entity, instead a *DimStyleOverride* object is returned, the dimension entity is stored as *dim.dimension*.

Placing Measurement Text

There are different predefined DIMSTYLES to achieve various text placing locations.

DIMSTYLE 'EZ_RADIUS' settings are: 1 drawing unit is 1m, scale 1:100, length_factor is 100 which creates measurement text in cm, and a closed filled arrow with size 0.25 is used.

**Note:** Not all possible features of DIMSTYLE are supported and especially for diameter dimension there are less features supported as for linear dimension because of the lack of good documentation.

See also:

- Graphical reference of many DIMVARS and some advanced information: *DIMSTYLE Table*
- Source code file *standards.py* shows how to create your own DIMSTYLES.
- *dimension_diameter.py* for diameter dimension examples.
**Default Text Locations Outside**

'**EZ_RADIUS**' default settings for to place text outside:

<table>
<thead>
<tr>
<th>tmove</th>
<th>1 to keep dim line with text, this is the best setting for text outside to preserve appearance of the DIMENSION entity, if editing afterwards in BricsCAD or AutoCAD.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dimtad</td>
<td>1 to place text vertical above the dimension line</td>
</tr>
</tbody>
</table>

```python
dim = msp.add_diameter_dim(center=(0, 0), radius=2.5, angle=45,
    dimstyle='EZ_RADIUS')
dim.render()  # required, but not shown in the following examples
```

To force text outside horizontal set **dimtoh** to 1:

```python
dim = msp.add_diameter_dim(center=(0, 0), radius=2.5, angle=45,
    dimstyle='EZ_RADIUS',
    override={'dimtoh': 1})
```

**Default Text Locations Inside**

**DIMSTYLE 'EZ_RADIUS_INSIDE'** can be used to place the dimension text inside the circle at a default location. Default DIMSTYLE settings are: 1 drawing unit is 1m, scale 1:100, length_factor is 100 which creates measurement text in cm, and a closed filled arrow with size 0.25 is used.
'EZ_RADIUS_INSIDE' default settings:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmove</td>
<td>0</td>
<td>to keep dim line with text, this is the best setting for text inside to preserve appearance of the DIMENSION entity, if editing afterwards in BricsCAD or AutoCAD.</td>
</tr>
<tr>
<td>dimtix</td>
<td>1</td>
<td>to force text inside</td>
</tr>
<tr>
<td>dimmatfit</td>
<td>0</td>
<td>to force text inside, required by BricsCAD and AutoCAD</td>
</tr>
<tr>
<td>dimtad</td>
<td>0</td>
<td>to center text vertical, BricsCAD and AutoCAD always create vertical centered text. ezdxr let you choose the vertical placement (above, below, center), but editing the DIMENSION in BricsCAD or AutoCAD will reset text to center placement.</td>
</tr>
</tbody>
</table>

```python
dim = msp.add_diameter_dim(center=(0, 0), radius=2.5, angle=45, 
                          dimstyle='EZ_RADIUS_INSIDE', 
                          override={'dimtad': 1})
```

To force text inside horizontal set `dimtih` to 1:

```python
dim = msp.add_diameter_dim(center=(0, 0), radius=2.5, angle=45, 
                          dimstyle='EZ_RADIUS_INSIDE', 
                          override={'dimtih': 1})
```
User Defined Text Locations

Beside the default location it is always possible to override the text location by a user defined location. This location also determines the angle of the dimension line and overrides the argument angle. For user defined locations it is not necessary to force text inside (dimtix=1), because the location of the text is explicit given, therefore the DIMSTYLE 'EZ_RADIUS' can be used for all this examples.

User defined location outside of the circle:

```python
dim = msp.add_diameter_dim(center=(0, 0), radius=2.5, location=(4, 4),
                           dimstyle='EZ_RADIUS')
```
User defined location outside of the circle and forced horizontal text:

```python
dim = msp.add_diameter_dim(center=(0, 0), radius=2.5, location=(4, 4),
    dimstyle='EZ_RADIUS',
    override={'dimtoh': 1})
```

User defined location inside of the circle:

```python
dim = msp.add_diameter_dim(center=(0, 0), radius=2.5, location=(1, 1),
    dimstyle='EZ_RADIUS')
```

User defined location inside of the circle and forced horizontal text:

```python
dim = msp.add_diameter_dim(center=(0, 0), radius=2.5, location=(1, 1),
    dimstyle='EZ_RADIUS',
    override={'dimtih': 1})
```
Center Mark/Lines

See Radius Dimension Tutorial: Center Mark/Lines

Overriding Measurement Text

See Linear Dimension Tutorial: Overriding Measurement Text

Measurement Text Formatting and Styling

See Linear Dimension Tutorial: Measurement Text Formatting and Styling

6.5.22 Tutorial for the Geo Add-on

This tutorial shows how to load a GPS track into a geo located DXF file and also the inverse operation, exporting geo located DXF entities as GeoJSON files.

Please read the section Intended Usage in the documentation of the ezdxf.addons.geo module first.

Warning: TO ALL BEGINNERS!

If you are just learning to work with geospatial data, using DXF files is not the way to go! DXF is not the first choice for storing data for spatial data analysts. If you run into problems I cannot help you as I am just learning myself.

The complete source code and test data for this tutorial are available in the github repository:

https://github.com/mozman/ezdxf/tree/master/docs/source/tutorials/src/geo

Setup Geo Location Reference

The first step is setting up the geo location reference, which is not doable with ezdxf yet - this feature may come in the future - but for now you have to use a CAD application to do this. If the DXF file has no geo location reference the projected 2D coordinates are most likely far away from the WCS origin (0, 0), use the CAD command “ZOOM EXTENDS” to find the data.
Load GPX Data

The GPX format stores GPS data in a XML format, use the `ElementTree` class to load the data:

```python
def load_gpx_track(p: Path) -> Iterable[Tuple[float, float]]:
    """ Load all track points from all track segments at once. """
    gpx = ET.parse(p)
    root = gpx.getroot()
    for track_point in root.findall('.//gpx:trkpt', GPX_NS):
        data = track_point.attrib
        # Elevation is not supported by the geo add-on.
        yield float(data['lon']), float(data['lat'])
```

The loaded GPS data has a WSG84 EPSG:4326 projection as longitude and latitude in decimal degrees. The next step is to create a `GeoProxy` object from this data, the `GeoProxy.parse()` method accepts a `__geo_interface__` mapping or a Python object with a `__geo_interface__` attribute/property. In this case as simple “LineString” object for all GPS points is sufficient:

```python
def add_gpx_track(msp, track_data, layer: str):
    geo_mapping = {
        'type': 'LineString',
        'coordinates': track_data,
    }
    geo_track = geo.GeoProxy.parse(geo_mapping)
```

Transform the data from the polar representation EPSG:4326 into a 2D cartesian map representation EPSG:3395 called “World Mercator”, this is the only projection supported by the add-on, without the need to write a custom transformation function:

```python
geo_track.globe_to_map()
```

The data is now transformed into 2D cartesian coordinates in meters and most likely far away from origin (0, 0), the data stored in the GEODATA entity helps to transform the data into the DXF WCS in modelspace units, if the DXF file has no geo location reference you have to stick with the large coordinates:

```python
# Load geo data information from the DXF file:
geo_data = msp.get_geodata()
if geo_data:
    # Get the transformation matrix and epsg code:
    m, epsg = geo_data.get_crs_transformation()
else:
    # Identity matrix for DXF files without a geo location reference:
    m = Matrix44()
    epsg = 3395
# Check for compatible projection:
if epsg == 3395:
    # Transform CRS coordinates into DXF WCS:
    geo_track.crs_to_wcs(m)
    # Create DXF entities (LWPOLYLINE)
    for entity in geo_track.to_dxf_entities(dxfattribs={'layer': layer}):
        # Add entity to the modelspace:
        msp.add_entity(entity)
else:
    print(f'Incompatible CRS EPSG: {epsg}')
```

We are ready to save the final DXF file:
In BricsCAD the result looks like this, the underlying images were added by the BricsCAD command MAPCONNECT and such a feature is not planned for the add-on:

Export DXF Entities as GeoJSON

This will only work with a proper geo location reference, the code shown accepts also WCS data from DXF files without a GEODATA object, but the result is just unusable - but in valid GeoJSON notation.

First get epsg code and the CRS transformation matrix:

```
# Get the geo location information from the DXF file:
geo_data = msp.get_geodata()
if geo_data:
    # Get transformation matrix and epsg code:
    m, epsg = geo_data.get_crs_transformation()
else:
```

(continues on next page)
# Identity matrix for DXF files without geo reference data:

```python
m = Matrix44()
```

Query the DXF entities to export:

```python
m = Matrix44()
for track in msp.query('LWPOLYLINE'):
```

Create a GeoProxy object from the DXF entity:

```python
def export_geojson(entity, m):
    # Convert DXF entity into a GeoProxy object:
    geo_proxy = geo.proxy(entity)
```

Transform DXF WCS coordinates in modelspace units into the CRS coordinate system by the transformation matrix \( m \):

```python
# Transform DXF WCS coordinates into CRS coordinates:
geo_proxy.wcs_to_crs(m)
```

The next step assumes a EPSG:3395 projection, everything else needs a custom transformation function:

```python
# Transform 2D map projection EPSG:3395 into globe (polar)
# representation EPSG:4326
geo_proxy.map_to_globe()
```

Use the `json` module from the Python standard library to write the GeoJSON data, provided by the GeoProxy. `__geo_interface__` property:

```python
# Export GeoJSON data:
name = entity.dxf.layer + '.geojson'
with open(TRACK_DATA / name, 'wt', encoding='utf8') as fp:
    json.dump(geo_proxy.__geo_interface__, fp, indent=2)
```

The content of the GeoJSON file looks like this:

```json
{
    "type": "LineString",
    "coordinates": [
        [15.430999, 47.06503],
        [15.431039, 47.064797],
        [15.431206, 47.064582],
        [15.431283, 47.064342],
        ...
    ]
}
```
Custom Transformation Function

This section shows how to use the GDAL package to write a custom transformation function. The example reimplements the built-in transformation from unprojected WGS84 coordinates to 2D map coordinates EPSG:3395 “World Mercator”:

```python
from osgeo import osr
from ezdxf.math import Vec3

gpx_points = list(load_gpx_track('track1.gpx'))

crsrc = osr.SpatialReference()
crsrc.SetWellKnownGeogCS('WGS84')

target_datum = osr.SpatialReference()
target_datum.SetWellKnownGeogCS('EPSG:3395')

c = osr.CoordinateTransform(src_datum, target_datum)

g = GeoProxy.parse({
    'type': 'LineString',
    'coordinates': gpx_points
})

g.apply(lambda v: Vec3(c.TransformPoint(v.x, v.y)))
```

The same example with the pyproj package:

```python
from pyproj import Transformer
from ezdxf.math import Vec3

gpx_points = list(load_gpx_track('track1.gpx'))

c = Transformer.from_crs('EPSG:4326', 'EPSG:3395')

g = GeoProxy.parse({
    'type': 'LineString',
    'coordinates': gpx_points
})

g.apply(lambda v: Vec3(c.transform(v.x, v.y)))
```

Polygon Validation by Shapely

Ezdxf tries to avoid to create invalid polygons from HATCH entities like a hole in another hole, but not all problems are detected by ezdxf, especially overlapping polygons. For a reliable and robust result use the Shapely package to check for valid polygons:
import ezdxf
from ezdxf.addons import geo
from shapely.geometry import shape

# Load DXF document including HATCH entities.
doc = ezdxf.readfile('hatch.dxf')
msp = doc.modelspace()

# Test a single entity
# Get the first DXF hatch entity:
hatch_entity = msp.query('HATCH').first

# Create GeoProxy() object:
hatch_proxy = geo.proxy(hatch_entity)

# Shapely supports the __geo_interface__
shapley_polygon = shape(hatch_proxy)

if shapely_polygon.is_valid:
    ...
else:
    print(f'Invalid Polygon from {str(hatch_entity)}.

# Remove invalid entities by a filter function
def validate(geo_proxy: geo.GeoProxy) -> bool:
    # Multi-entities are divided into single entities:
    # e.g. MultiPolygon is verified as multiple single Polygon entities.
    if geo_proxy.geotype == 'Polygon':
        return shape(geo_proxy).is_valid
    return True

# The gfilter() function let only pass compatible DXF entities
msp_proxy = geo.GeoProxy.from_dxf_entities(geo.gfilter(msp))

# remove all mappings for which validate() returns False
msp_proxy.filter(validate)

---

Interface to GDAL/OGR

The GDAL/OGR package has no direct support for the __geo_interface__, but has built-in support for the GeoJSON format:

```python
from osgeo import ogr
from ezdxf.addons import geo
from ezdxf.render import random_2d_path
import json

p = geo.GeoProxy({'type': 'LineString', 'coordinates': list(random_2d_path(20))})
# Create a GeoJSON string from the __geo_interface__ object by the json
# module and feed the result into ogr:
line_string = ogr.CreateGeometryFromJson(json.dumps(p.__geo_interface__))

# Parse the GeoJSON string from ogr by the json module and feed the result
# into a GeoProxy() object:
p2 = geo.GeoProxy.parse(json.loads(line_string.ExportToJson()))
```
6.6 Howto

The Howto section show how to accomplish specific tasks with ezdxf in a straight forward way without teaching basics or internals, if you are looking for more information about the ezdxf internals look at the Reference section or if you want to learn how to use ezdxf go to the Tutorials section or to the Basic Concepts section.

6.6.1 General Document

General preconditions:

```python
import sys
import ezdxf

try:
    doc = ezdxf.readfile("your_dxf_file.dxf")
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file.')
sys.exit(2)
msp = doc.modelspace()
```

This works well with DXF files from trusted sources like AutoCAD or BricsCAD, for loading DXF files with minor or major flaws look at the ezdxf.recover module.

Load DXF Files with Structure Errors

If you know the files you will process have most likely minor or major flaws, use the ezdxf.recover module:

```python
import sys
from ezdxf import recover

try:  # low level structure repair:
    doc, auditor = recover.readfile(name)
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file: {name}.
    sys.exit(2)

# DXF file can still have unrecoverable errors, but this is maybe just a problem when saving the recovered DXF file.
if auditor.has_errors:
    print(f'Found unrecoverable errors in DXF file: {name}.
    auditor.print_error_report()
```

For more loading scenarios follow the link: ezdxf.recover

Set/Get Header Variables

ezdxf has an interface to get and set HEADER variables:
See also:

HeaderTextSection and online documentation from Autodesk for available header variables.

Set DXF Drawing Units

The header variable $INSUNITS defines the drawing units for the modelspace and therefore for the DXF document if no further settings are applied. The most common units are 6 for meters and 1 for inches.

Use this HEADER variables to setup the default units for CAD applications opening the DXF file. This setting is not relevant for ezdxf API calls, which are unitless for length values and coordinates and decimal degrees for angles (in most cases).

Sets drawing units:

```py
doc.header['$INSUNITS'] = 6
```

For more information see section DXF Units.

Create More Readable DXF Files (DXF Pretty Printer)

DXF files are plain text files, you can open this files with every text editor which handles bigger files. But it is not really easy to get quick the information you want.

Create a more readable HTML file (DXF Pretty Printer):

This creates a HTML file with a nicer layout than a plain text file, and handles are links between DXF entities, this simplifies the navigation between the DXF entities.

Changed in version 0.16: The dxfpp command was replaced by a sub-command of the ezdxf launcher.

```

positional arguments:
  FILE  DXF files pretty print

optional arguments:
  -h, --help             show this help message and exit
  -o, --open             open generated HTML file with the default web browser
  -r, --raw              raw mode - just print tags, no DXF structure interpretation
  -x, --nocompile        don't compile points coordinates into single tags (only in raw mode)
  -l, --legacy           legacy mode - reorders DXF point coordinates
```

**Important:** This does not render the graphical content of the DXF file to a HTML canvas element.

Calculate Extents for the Modelspace

Since ezdxf v0.16 exist a ezdxf.bbox module to calculate bounding boxes for DXF entities. This module makes the extents calculation very easy, but read the documentation for the bbox module to understand its limitations.
import ezdxf
from ezdxf import bbox

doc = ezdxf.readfile("your.dxf")
msp = doc.modelspace()
extents = bbox.extents(msp)

The returned *extents* is a *ezdxf.math.BoundingBox* object.

### Set Initial View/Zoom for the Modelspace

To show an arbitrary location of the modelspace centered in the CAD application window, set the '*Active' VPORT to this location. The DXF attribute *dxf.center* defines the location in the modelspace, and the *dxf.height* specifies the area of the modelspace to view. Shortcut function:

```python
doc.set_modelspace_vport(height=10, center=(10, 10))
```

New in version 0.16.

The new *ezdxf.zoom* module of *ezdxf* v0.16, makes this task much easier.

Setting the initial view to the extents of all entities in the modelspace:

```python
import ezdxf
from ezdxf import zoom
doc = ezdxf.readfile("your.dxf")
msp = doc.modelspace()
zoom.extents(msp)
```

Setting the initial view to the extents of just some entities:

```python
lines = msp.query("LINES")
zoom.objects(lines)
```

The *zoom* module also works for paperspace layouts.

**Important:** The *zoom* module uses the *bbox* module to calculate the bounding boxes for DXF entities. Read the documentation for the *bbox* module to understand its limitations and the bounding box calculation for large documents can take a while!

### 6.6.2 DXF Viewer

#### A360 Viewer Problems

AutoDesk web service *A360* seems to be more picky than the AutoCAD desktop applications, may be it helps to use the latest DXF version supported by ezdxf, which is DXF R2018 (AC1032) in the year of writing this lines (2018).

#### DXF Entities Are Not Displayed in the Viewer

*ezdxf* does not automatically locate the main viewport of the modelspace at the entities, you have to perform the “Zoom to Extends” command, here in TrueView 2020:

---

6.6. Howto 131
Add this line to your code to relocate the main viewport, adjust the center (in modelspace coordinates) and the height (in drawing units) arguments to your needs:

\[
doc.set_modelspace_vport(height=10, center=(0, 0))
\]

**Show IMAGES/XREFS on Loading in AutoCAD**

If you are adding XREFS and IMAGES with relative paths to existing drawings and they do not show up in AutoCAD immediately, change the HEADER variable $PROJECTNAME=' ' to *(not really)* solve this problem. The ezdxf templates for DXF R2004 and later have $PROJECTNAME=' ' as default value.
Thanks to David Booth:

If the filename in the IMAGEDEF contains the full path (absolute in AutoCAD) then it shows on loading, otherwise it won’t display (reports as unreadable) until you manually reload using XREF manager.

A workaround (to show IMAGES on loading) appears to be to save the full file path in the DXF or save it as a DWG.

So far - no solution for showing IMAGES with relative paths on loading.

Set Initial View/Zoom for the Modelspace

See section “General Document”: Set Initial View/Zoom for the Modelspace

6.6.3 DXF Content

General preconditions:

```python
import sys
import ezdxf

try:
    doc = ezdxf.readfile("your_dxf_file.dxf")
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file.')
sys.exit(2)
msp = doc.modelspace()
```

Get/Set Entity Color

The entity color is stored as ACI (AutoCAD Color Index):

```python
aci = entity.dxf.color
```

Default value is 256 which means BYLAYER:

```python
layer = doc.layers.get(entity.dxf.layer)
aci = layer.get_color()
```

The special `get_color()` method is required, because the color attribute `Layer.dxf.color` is misused as layer on/off flag, a negative color value means the layer is off.

ACI value 0 means BYBLOCK, which means the color from the block reference (INSERT entity).

Set color as ACI value as int in range [0, 256]:

```python
entity.dxf.color = 1
```

The RGB values of the AutoCAD default colors are not officially documented, but an accurate translation table is included in `ezdxf`:
from ezdxf.colors import DXF_DEFAULT_COLORS, int2rgb

# 24 bit value RRRRRRRGGGGGGGBBBBBB
rgb24 = DXF_DEFAULT_COLORS[aci]
print(f'RGB Hex Value: #{rgb24:06X}')

r, g, b = int2rgb(rgb24)
print(f'RGB Channel Values: R={r:02X} G={g:02X} b={b:02X}')

The ACI value 7 has a special meaning, it is white on dark backgrounds and white on light backgrounds.

**Get/Set Entity RGB Color**

RGB true color values are supported since DXF R13 (AC1012), the 24-bit RGB value is stored as integer in the DXF attribute `true_color`:

```python
# set true color value to red
entity.dxf.true_color = 0xFF0000
```

The `rgb` property of the `DXFGraphic` entity add support to get/set RGB value as (r, g, b)-tuple:

```python
# set true color value to red
entity.rgb = (255, 0, 0)
```

If `color` and `true_color` values are set, BricsCAD and AutoCAD use the `true_color` value as display color for the entity.

**Get/Set Block Reference Attributes**

Block references (`Insert`) can have attached attributes (`Attrib`), these are simple text annotations with an associated tag appended to the block reference.

Iterate over all appended attributes:

```python
# get all INSERT entities with entity.dxf.name == "Part12"
blockrefs = msp.query('INSERT[name=="Part12"]')
if len(blockrefs):
    entity = blockrefs[0]  # process first entity found
    for attrib in entity.attribs:
        if attrib.dxf.tag == "diameter":  # identify attribute by tag
            attrib.dxf.text = "17mm"  # change attribute content
```

Get attribute by tag:

```python
diameter = entity.get_attrib('diameter')
if diameter is not None:
    diameter.dxf.text = "17mm"
```

**Adding XDATA to Entities**

Adding XDATA as list of tuples (group code, value) by `set_xdata()`, overwrites data if already present:
For group code meaning see DXF reference section DXF Group Codes in Numerical Order Reference, valid group codes are in the range 1000 - 1071.

Method `get_xdata()` returns the extended data for an entity as `Tags` object.

### Get Overridden DIMSTYLE Values from DIMENSION

In general the `Dimension` styling and config attributes are stored in the `Dimstyle` entity, but every attribute can be overridden for each `DIMENSION` entity individually, get overwritten values by the `DimstyleOverride` object as shown in the following example:

```python
for dimension in msp.query('DIMENSION'):
    dimstyle_override = dimension.override()  # requires v0.12
    if dimtol:
        print(f'{str(dimension)} has tolerance values: ')
        dimtp = dimstyle_override['dimtp']
        dimtm = dimstyle_override['dimtm']
        print(f'Upper tolerance: {dimtp}')
        print(f'Lower tolerance: {dimtm}')
```

The `DimstyleOverride` object returns the value of the underlying `DIMSTYLE` objects if the value in `DIMENSION` was not overwritten, or `None` if the value was neither defined in `DIMSTYLE` nor in `DIMENSION`.

### Override DIMSTYLE Values for DIMENSION

Same as above, the `DimstyleOverride` object supports also overriding `DIMSTYLE` values. But just overriding these values have no effect on the graphical representation of the `DIMENSION` entity, because CAD applications just show the associated anonymous block which contains the graphical representation on the `DIMENSION` entity as simple DXF entities. Call the `render` method of the `DimstyleOverride` object to recreate this graphical representation by `ezdxf`, but `ezdxf` **does not** support all `DIMENSION` types and `DIMVARS` yet, and results will differ from AutoCAD or BricsCAD renderings.

```python
dimstyle_override = dimension.override()
dimstyle_override.set_tolerance(0.1)
# delete associated geometry block
del doc.blocks[dimension.dxf.geometry]
```
# recreate geometry block

dimstyle_override.render()

## 6.7 FAQ

### 6.7.1 What is the Relationship between ezdxf, dxfwrite and dxfgrabber?

In 2010 I started my first Python package for creating DXF documents called *dxfwrite*, this package can’t read DXF files and writes only the DXF R12 (AC1009) version. While *dxfwrite* works fine, I wanted a more versatile package, that can read and write DXF files and maybe also supports newer DXF formats than DXF R12.

This was the start of the *ezdxf* package in 2011, but the progress was so slow, that I created a spin off in 2012 called *dxfgrabber*, which implements only the reading part of *ezdxf*, which I needed for my work and I wasn’t sure if *ezdxf* will ever be usable. Luckily in 2014 the first usable version of *ezdxf* could be released. The *ezdxf* package has all the features of *dxfwrite* and *dxfgrabber* and much more, but with a different API. So *ezdxf* is not a drop-in replacement for *dxfgrabber* or *dxfwrite*.

Since *ezdxf* can do all the things that *dxfwrite* and *dxfgrabber* can do, I focused on the development of *ezdxf*, *dxfwrite* and *dxfgrabber* are in maintenance mode only and will not get any new features, just bugfixes.

There are no advantages of *dxfwrite* over *ezdxf*, *dxfwrite* has the smaller memory footprint, but the *r12writer* add-on does the same job as *dxfwrite* without any in memory structures by writing direct to a stream or file and there is also no advantage of *dxfgrabber* over *ezdxf* for normal DXF files the smaller memory footprint of *dxfgrabber* is not noticeable and for really big files the *iterdxf* add-on does a better job.

### 6.7.2 Imported ezdxf package has no content. (readfile, new)

1. AttributeError: partially initialized module 'ezdxf' has no attribute 'readfile' (most likely due to a circular import)

   Did you name your file/script “ezdxf.py”? This causes problems with circular imports. Renaming your file/script should solve this issue.

2. AttributeError: module 'ezdxf' has no attribute 'readfile'

   This could be a hidden permission error, for more information about this issue read Petr Zemek's article: [https://blog.petrzemek.net/2020/11/17/when-you-import-a-python-package-and-it-is-empty/](https://blog.petrzemek.net/2020/11/17/when-you-import-a-python-package-and-it-is-empty/)

## 6.8 Reference

The DXF Reference is online available at Autodesk.

Quoted from the original DXF 12 Reference which is not available on the web:

Since the AutoCAD drawing database (.dwg file) is written in a compact format that changes significantly as new features are added to AutoCAD, we do not document its format and do not recommend that you attempt to write programs to read it directly. To assist in interchanging drawings between AutoCAD and other programs, a Drawing Interchange file format (DXF) has been defined. All implementations of AutoCAD accept this format and are able to convert it to and from their internal drawing file representation.
6.8.1 DXF Document

Document Management

Create New Drawings

ezdxf.new(dxversion='AC1027', setup=False, units=6) → Drawing

Create a new Drawing from scratch, dxversion can be either “AC1009” the official DXF version name or “R12” the AutoCAD release name.

new() can create drawings for following DXF versions:

<table>
<thead>
<tr>
<th>Version</th>
<th>AutoCAD Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC1009</td>
<td>AutoCAD R12</td>
</tr>
<tr>
<td>AC1015</td>
<td>AutoCAD R2000</td>
</tr>
<tr>
<td>AC1018</td>
<td>AutoCAD R2004</td>
</tr>
<tr>
<td>AC1021</td>
<td>AutoCAD R2007</td>
</tr>
<tr>
<td>AC1024</td>
<td>AutoCAD R2010</td>
</tr>
<tr>
<td>AC1027</td>
<td>AutoCAD R2013</td>
</tr>
<tr>
<td>AC1032</td>
<td>AutoCAD R2018</td>
</tr>
</tbody>
</table>

The units argument defines the document and modelspace units. The header variable $MEASUREMENT will be set according to the given units, 0 for inch, feet, miles, . . . and 1 for metric units. For more information go to module ezdxf.units

Parameters

- dxversion – DXF version specifier as string, default is “AC1027” respectively “R2013”
- setup – setup default styles, False for no setup, True to setup everything or a list of topics as strings, e.g. ["linetypes", "styles"] to setup only some topics:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linetypes</td>
<td>setup line types</td>
</tr>
<tr>
<td>styles</td>
<td>setup text styles</td>
</tr>
<tr>
<td>dimstyles</td>
<td>setup default ezdxf dimension styles</td>
</tr>
<tr>
<td>visualstyles</td>
<td>setup 25 standard visual styles</td>
</tr>
</tbody>
</table>

- units – document and modelspace units, default is 6 for meters

Open Drawings

Open DXF drawings from file system or text stream, byte stream usage is not supported.

DXF files prior to R2007 requires file encoding defined by header variable $DWGCODEPAGE, DXF R2007 and later requires an UTF-8 encoding.

ezdxf supports reading of files for following DXF versions:
ezdxf.readfile (filename: str, encoding: str = None, errors: str = "surrogateescape") → Drawing

Read the DXF document filename from the file-system.

This is the preferred method to load existing ASCII or Binary DXF files, the required text encoding will be detected automatically and decoding errors will be ignored.

Override encoding detection by setting argument encoding to the estimated encoding. (use Python encoding names like in the open() function).

If this function struggles to load the DXF document and raises a DXFStructureError exception, try the ezdxf.recover.readfile() function to load this corrupt DXF document.

Parameters

- **filename** – filename of the ASCII- or Binary DXF document
- **encoding** – use None for auto detect (default), or set a specific encoding like “utf-8”, argument is ignored for Binary DXF files
- **errors** – specify decoding error handler
  - “surrogateescape” to preserve possible binary data (default)
  - “ignore” to use the replacement char U+FFFD “” for invalid data
  - “strict” to raise an UnicodeDecodeError exception for invalid data

Raises

- IOError – not a DXF file or file does not exist
- DXFStructureError – for invalid or corrupted DXF structures
- UnicodeDecodeError – if errors is “strict” and a decoding error occurs

Deprecated since version v0.14: argument legacy_mode, use module ezdxf.recover to load DXF documents with structural flaws.

ezdxf.read (stream: TextIO) → Drawing

Read a DXF document from a text-stream. Open stream in text mode (mode='rt') and set correct text encoding, the stream requires at least a readline() method.

Since DXF version R2007 (AC1021) file encoding is always “utf-8”, use the helper function dxf_stream_info() to detect the required text encoding for prior DXF versions. To preserve possible binary data in use errors='surrogateescape' as error handler for the import stream.

If this function struggles to load the DXF document and raises a DXFStructureError exception, try the ezdxf.recover.read() function to load this corrupt DXF document.

Parameters stream – input text stream opened with correct encoding
Raises DXFStructureError – for invalid or corrupted DXF structures

Deprecated since version v0.14: argument legacy_mode, use module ezdxf.recover to load DXF documents with structural flaws.

ezdxf.readzip(zipfile: str, filename: str = None, errors: str="surrogateescape") → Drawing

Load a DXF document specified by filename from a zip archive, or if filename is None the first DXF document in the zip archive.

Parameters

• zipfile – name of the zip archive
• filename – filename of DXF file, or None to load the first DXF document from the zip archive.
• errors – specify decoding error handler
  – ”surrogateescape” to preserve possible binary data (default)
  – ”ignore” to use the replacement char U+FFFD “” for invalid data
  – ”strict” to raise an UnicodeDecodeError exception for invalid data

Raises

• IOError – not a DXF file or file does not exist or if filename is None - no DXF file found
• DXFStructureError – for invalid or corrupted DXF structures
• UnicodeDecodeError – if errors is “strict” and a decoding error occurs

ezdxf.decode_base64(data: bytes, errors: str="surrogateescape") → Drawing

Load a DXF document from base64 encoded binary data, like uploaded data to web applications.

Parameters

• data – DXF document base64 encoded binary data
• errors – specify decoding error handler
  – ”surrogateescape” to preserve possible binary data (default)
  – ”ignore” to use the replacement char U+FFFD “” for invalid data
  – ”strict” to raise an UnicodeDecodeError exception for invalid data

Raises

• DXFStructureError – for invalid or corrupted DXF structures
• UnicodeDecodeError – if errors is “strict” and a decoding error occurs

Hint: This works well with DXF files from trusted sources like AutoCAD or BricsCAD, for loading DXF files with minor or major flaws look at the ezdxf.recover module.

Save Drawings

Save the DXF document to the file system by Drawing methods save() or saveas(). Write the DXF document to a text stream with write(), the text stream requires at least a write() method. Get required output encoding for text streams by property Drawing.output_encoding
**Drawing Settings**

The *HeaderSection* stores meta data like modelspace extensions, user name or saving time and current application settings, like actual layer, text style or dimension style settings. These settings are not necessary to process DXF data and therefore many of this settings are not maintained by *ezdxf* automatically.

**Header variables set at new**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ACADVER</td>
<td>DXF version</td>
</tr>
<tr>
<td>$TDCREATE</td>
<td>date/time at creating the drawing</td>
</tr>
<tr>
<td>$FINGERPRINTGUID</td>
<td>every drawing gets a GUID</td>
</tr>
</tbody>
</table>

**Header variables updated at saving**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$TDUPDATE</td>
<td>actual date/time saving</td>
</tr>
<tr>
<td>$HANDSEED</td>
<td>next available handle as hex string</td>
</tr>
<tr>
<td>$DWGCODEPAGE</td>
<td>encoding setting</td>
</tr>
<tr>
<td>$VERSIONGUID</td>
<td>every saved version gets a new GUID</td>
</tr>
</tbody>
</table>

See also:
- Howto: *Set/Get Header Variables*
- Howto: *Set DXF Drawing Units*

**Drawing Object**

The *Drawing* class manages all entities and tables related to a DXF drawing.

```
class ezdxf.document.Drawing
```

- **dxfversion**
  - Actual DXF version like 'AC1009', set by *ezdxf.new()* or *ezdxf.readfile()*.
  - For supported DXF versions see *Document Management*.

- **acad_release**
  - The AutoCAD release name like 'R12' or 'R2000' for actual *dxfversion*.

- **encoding**
  - Text encoding of *Drawing*, the default encoding for new drawings is 'cp1252'. Starting with DXF R2007 (AC1021), DXF files are written as UTF-8 encoded text files, regardless of the attribute *encoding*. Text encoding can be changed to encodings listed below.
  - see also: *DXF File Encoding*
supported | encodings
--- | ---
'cp874' | Thai
'cp932' | Japanese
'gbk' | UnifiedChinese
'cp949' | Korean
'cp950' | TradChinese
'cp1250' | CentralEurope
'cp1251' | Cyrillic
'cp1252' | WesternEurope
'cp1253' | Greek
'cp1254' | Turkish
'cp1255' | Hebrew
'cp1256' | Arabic
'cp1257' | Baltic
'cp1258' | Vietnam

output_encoding
Returns required output encoding for saving to filesystem or encoding to binary data.

filename
Drawing filename, if loaded by ezdxf.readfile() else None.

rootdict
Reference to the root dictionary of the OBJECTS section.

header
Reference to the HeaderSection, get/set drawing settings as header variables.

entities
Reference to the EntitySection of the drawing, where all graphical entities are stored, but only from modelspace and the active paperspace layout. Just for your information: Entities of other paperspace layouts are stored as BlockLayout in the BlocksSection.

objects
Reference to the objects section, see also ObjectsSection.

blocks
Reference to the blocks section, see also BlocksSection.

tables
Reference to the tables section, see also TablesSection.

classes
Reference to the classes section, see also ClassesSection.

layouts
Reference to the layout manager, see also Layouts.

groups
Collection of all groups, see also GroupCollection.

requires DXF R13 or later

layers
Shortcut for Drawing.tables.layers
Reference to the layers table, where you can create, get and remove layers, see also Table and Layer
styles
  Shortcut for `Drawing.tables.styles`
  Reference to the styles table, see also Style.

dimstyles
  Shortcut for `Drawing.tables.dimstyles`
  Reference to the dimstyles table, see also DimStyle.

linetypes
  Shortcut for `Drawing.tables.linetypes`
  Reference to the linetypes table, see also Linetype.

views
  Shortcut for `Drawing.tables.views`
  Reference to the views table, see also View.

viewports
  Shortcut for `Drawing.tables.viewports`
  Reference to the viewports table, see also Viewport.

ucs
  Shortcut for `Drawing.tables.ucs`
  Reference to the ucs table, see also UCS.

appids
  Shortcut for `Drawing.tables.appids`
  Reference to the appids table, see also AppID.

materials
  MaterialCollection of all Material objects.

mline_styles
  MLineStyleCollection of all MLineStyle objects.

mleader_styles
  MLeaderStyleCollection of all MLeaderStyle objects.

units
  Get and set the document/modelspace base units as enum, for more information read this: DXF Units.

save (encoding: str = None, fmt: str = 'asc') → None
  Write drawing to file-system by using the filename attribute as filename. Override file encoding by argument encoding, handle with care, but this option allows you to create DXF files for applications that handles file encoding different than AutoCAD.

Parameters
  • encoding – override default encoding as Python encoding string like 'utf-8'
  • fmt – 'asc' for ASCII DXF (default) or 'bin' for Binary DXF

saveas (filename: Union[str, Path], encoding: str = None, fmt: str = 'asc') → None
  Set Drawing attribute filename to filename and write drawing to the file system. Override file encoding by argument encoding, handle with care, but this option allows you to create DXF files for applications that handles file encoding different than AutoCAD.

Parameters
  • filename – file name as string
• **encoding** – override default encoding as Python encoding string like 'utf-8'
• **fmt** – 'asc' for ASCII DXF (default) or 'bin' for Binary DXF

```python
def write(stream: Union[TextIO, BinaryIO], fmt: str = 'asc') -> None
    Write drawing as ASCII DXF to a text stream or as Binary DXF to a binary stream. For DXF R2004 (AC1018) and prior open stream with drawing encoding and mode='wt'. For DXF R2007 (AC1021) and later use encoding='utf-8', or better use the later added Drawing property output_encoding which returns the correct encoding automatically. The correct and required error handler is errors='dxfreplace'!
    If writing to a StringIO stream, use Drawing.encode() to encode the result string from StringIO.get_value():
    ```
    ```
    binary = doc.encode(stream.get_value())
    ```

**Parameters**

- **stream** – output text stream or binary stream
- **fmt** – 'asc' for ASCII DXF (default) or 'bin' for Binary DXF

```python
def encode_base64() -> bytes
    Returns DXF document as base64 encoded binary data.
```

```python
def encode(s: str) -> bytes
    Encode string s with correct encoding and error handler.
```

```python
def query(query: str = '*') -> ezdxf.query.EntityQuery
    Entity query over all layouts and blocks, excluding the OBJECTS section.
    ```
    ```
    Parameters **query** – query string
    ```

See also:

- Entity Query String and Retrieve entities by query language

```python
def groupby(dxfattrib='', key=None) -> dict
    Groups DXF entities of all layouts and blocks (excluding the OBJECTS section) by a DXF attribute or a key function.
    ```
    ```
    Parameters
    - **dxfattrib** – grouping DXF attribute like 'layer'
    - **key** – key function, which accepts a DXFEntity as argument and returns a hashable grouping key or None to ignore this entity.
    ```

See also:

- groupby() documentation

```python
def modelspace() -> ezdxf.layouts.layout.Modelspace
    Returns the modelspace layout, displayed as 'Model' tab in CAD applications, defined by block record named '*Model_Space'.
```

```python
def layout(name: str = None) -> Layout
    Returns paperspace layout name or returns first layout in tab order if name is None.
```

```python
def active_layout() -> Layout
    Returns the active paperspace layout, defined by block record name '*Paper_Space'.
```

```python
def layout_names() -> Iterable[str]
    Returns all layout names (modelspace 'Model' included) in arbitrary order.
```
layout_names_in_taborder() → Iterable[str]
Returns all layout names (modelspace included, always first name) in tab order.

new_layout(name, dxfattribs=None) → Layout
Create a new paperspace layout name. Returns a Layout object. DXF R12 (AC1009) supports only one paperspace layout, only the active paperspace layout is saved, other layouts are dismissed.

Parameters

• name – unique layout name

• dxfattribs – additional DXF attributes for the DXFLayout entity

Raises DXFValueError – Layout name already exist

delete_layout(name: str) → None
Delete paper space layout name and all entities owned by this layout. Available only for DXF R2000 or later, DXF R12 supports only one paperspace and it can’t be deleted.

add_image_def(filename: str, size_in_pixel: Tuple[int, int], name=None)
Add an image definition to the objects section.
Add an ImageDef entity to the drawing (objects section). filename is the image file name as relative or absolute path and size_in_pixel is the image size in pixel as (x, y) tuple. To avoid dependencies to external packages, ezdxf can not determine the image size by itself. Returns a ImageDef entity which is needed to create an image reference. name is the internal image name, if set to None, name is auto-generated.

Absolute image paths works best for AutoCAD but not really good, you have to update external references manually in AutoCAD, which is not possible in TrueView. If the drawing units differ from 1 meter, you also have to use: set_raster_variables().

Parameters

• filename – image file name (absolute path works best for AutoCAD)

• size_in_pixel – image size in pixel as (x, y) tuple

• name – image name for internal use, None for using filename as name (best for AutoCAD)

See also:
Tutorial for Image and ImageDef

set_raster_variables(frame: int = 0, quality: int = 1, units: str = 'm')
Set raster variables.

Parameters

• frame – 0 = do not show image frame; 1 = show image frame

• quality – 0 = draft; 1 = high

• units – units for inserting images. This defines the real world unit for one drawing unit for the purpose of inserting and scaling images with an associated resolution.

<table>
<thead>
<tr>
<th>mm</th>
<th>Millimeter</th>
</tr>
</thead>
<tbody>
<tr>
<td>cm</td>
<td>Centimeter</td>
</tr>
<tr>
<td>m</td>
<td>Meter (ezdxf default)</td>
</tr>
<tr>
<td>km</td>
<td>Kilometer</td>
</tr>
<tr>
<td>in</td>
<td>Inch</td>
</tr>
<tr>
<td>ft</td>
<td>Foot</td>
</tr>
<tr>
<td>yd</td>
<td>Yard</td>
</tr>
<tr>
<td>mi</td>
<td>Mile</td>
</tr>
</tbody>
</table>
set_wipeout_variables(frame=0)
Set wipeout variables.

Parameters
frame – 0 = do not show image frame; 1 = show image frame

add_underlay_def(filename: str, format: str = 'ext', name: str = None)
Add an UnderlayDef entity to the drawing (OBJECTS section). filename is the underlay file name as relative or absolute path and format as string (pdf, dwf, dgn). The underlay definition is required to create an underlay reference.

Parameters

• filename – underlay file name
• format – file format as string 'pdf'|'dwf'|'dgn' or 'ext' for getting file format from filename extension
• name – pdf format = page number to display; dgn format = 'default'; dwf: ????

See also:
Tutorial for Underlay and UnderlayDefinition

add_xref_def(filename: str, name: str, flags: int = 20)
Add an external reference (xref) definition to the blocks section.

Parameters

• filename – external reference filename
• name – name of the xref block
• flags – block flags

layouts_and_blocks() → Iterable[GenericLayoutType]
Iterate over all layouts (modelspace and paperspace) and all block definitions.

chain_layouts_and_blocks() → Iterable[DXFEntity]
Chain entity spaces of all layouts and blocks. Yields an iterator for all entities in all layouts and blocks.

reset_fingerprint_guid()
Reset fingerprint GUID.

reset_version_guid()
Reset version GUID.

set_modelspace_vport(height, center=(0, 0)) → VPort
Set initial view/zoom location for the modelspace, this replaces the current “*Active” viewport configuration.

Parameters

• height – modelspace area to view
• center – modelspace location to view in the center of the CAD application window.

audit() → Auditor
Checks document integrity and fixes all fixable problems, not fixable problems are stored in Auditor.errors.

If you are messing around with internal structures, call this method before saving to be sure to export valid DXF documents, but be aware this is a long running task.

validate(print_report=True) → bool
Simple way to run an audit process. Fixes all fixable problems, return False if not fixable errors occurs, to get more information about not fixable errors use audit() method instead.
Parameters **print_report** – print report to stdout

Returns: **True** if no errors occurred

**Recover**

New in version v0.14.

This module provides functions to “recover” ASCII DXF documents with structural flaws, which prevents the regular `ezdxf.read()` and `ezdxf.readfile()` functions to load the document.

The `read()` and `readfile()` functions will repair as much flaws as possible and run the required audit process automatically afterwards and return the result of this audit process:

```python
import sys
import ezdxf
from ezdxf import recover

try:
    doc, auditor = recover.readfile("messy.dxf")
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file.')
sys.exit(2)

# DXF file can still have unrecoverable errors, but this is maybe just
# a problem when saving the recovered DXF file.
if auditor.has_errors:
    auditor.print_error_report()
```

This efforts cost some time, loading the DXF document with `ezdxf.read()` or `ezdxf.readfile()` will be faster.

**Warning:** This module will load DXF files which have decoding errors, most likely binary data stored in XRECORD entities, these errors are logged as unrecoverable AuditError.DECODE_ERRORS in the Auditor.errors attribute, but no DXFStructureError exception will be raised, because for many use cases this errors can be ignored.

Writing such files back with `ezdxf` may create **invalid** DXF files, or at least some **information will be lost** - handle with care!

To avoid this problem use `recover.readfile(filename, errors='strict')` which raises an UnicodeDecodeError exception for such binary data. Catch the exception and handle this DXF files as unrecoverable.

**Loading Scenarios**

1. **It will work**

Mostly DXF files from AutoCAD or BricsCAD (e.g. for In-house solutions):
try:
    doc = ezdxf.readfile(name)
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file: {name}.
    sys.exit(2)

2. DXF file with minor flaws

DXF files have only minor flaws, like undefined resources:

try:
    doc = ezdxf.readfile(name)
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file: {name}.
    sys.exit(2)
auditor = doc.audit()
if auditor.has_errors:
    auditor.print_error_report()

3. Try Hard

From trusted and untrusted sources but with good hopes, the worst case works like a cache miss, you pay for the first try and pay the extra fee for the recover mode:

try:  # Fast path:
    doc = ezdxf.readfile(name)
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
# Catch all DXF errors:
except ezdxf.DXFError:
    try:  # Slow path including fixing low level structures:
        doc, auditor = recover.readfile(name)
    except ezdxf.DXFStructureError:
        print(f'Invalid or corrupted DXF file: {name}.
        sys.exit(2)
# DXF file can still have unrecoverable errors, but this is maybe
# just a problem when saving the recovered DXF file.
if auditor.has_errors:
    print(f'Found unrecoverable errors in DXF file: {name}.
    auditor.print_error_report()
4. Just use the slow recover module

Untrusted sources and expecting many invalid or corrupted DXF files, you always pay an extra fee for the recover mode:

```python
try:  # Slow path including fixing low level structures:
    doc, auditor = recover.readfile(name)
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file: {name}.')
    sys.exit(2)

# DXF file can still have unrecoverable errors, but this is maybe
# just a problem when saving the recovered DXF file.
if auditor.has_errors:
    print(f'Found unrecoverable errors in DXF file: {name}.')
    auditor.print_error_report()
```

5. Unrecoverable Decoding Errors

If files contain binary data which can not be decoded by the document encoding, it is maybe the best to ignore this files, this works in normal and recover mode:

```python
try:
    doc, auditor = recover.readfile(name, errors='strict')
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file: {name}.')
    sys.exit(2)
except UnicodeDecodeError:
    print(f'Decoding error in DXF file: {name}.')
    sys.exit(3)
```

6. Ignore/Locate Decoding Errors

Sometimes ignoring decoding errors can recover DXF files or at least you can detect where the decoding errors occur:

```python
try:
    doc, auditor = recover.readfile(name, errors='ignore')
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file: {name}.')
    sys.exit(2)
if auditor.has_errors:
    auditor.print_report()
```

The error messages with code AuditError.DECODING_ERROR shows the approximate line number of the decoding error: “Fixed unicode decoding error near line: xxx.”
ezdxf Documentation, Release 0.16.2

**Hint:** This function can handle only ASCII DXF files!

```python
ezdxf.recover.readfile(filename: str, errors: str = 'surrogateescape') \rightarrow Tuple[Drawing, Auditor]
```

Read a DXF document from file system similar to `ezdxf.readfile()`, but this function will repair as much flaws as possible, runs the required audit process automatically the DXF document and the Auditor.

**Parameters**

- **filename** – file-system name of the DXF document to load
- **errors** – specify decoding error handler
  - "surrogateescape" to preserve possible binary data (default)
  - "ignore" to use the replacement char U+FFFD "" for invalid data
  - "strict" to raise an UnicodeDecodeError exception for invalid data

**Raises**

- DXFStructureError – for invalid or corrupted DXF structures
- UnicodeDecodeError – if errors is “strict” and a decoding error occurs

```python
ezdxf.recover.read(stream: BinaryIO, errors: str = 'surrogateescape') \rightarrow Tuple[Drawing, Auditor]
```

Read a DXF document from a binary-stream similar to `ezdxf.read()`, but this function will detect the text encoding automatically and repair as much flaws as possible, runs the required audit process afterwards and returns the DXF document and the Auditor.

**Parameters**

- **stream** – data stream to load in binary read mode
- **errors** – specify decoding error handler
  - "surrogateescape" to preserve possible binary data (default)
  - "ignore" to use the replacement char U+FFFD "" for invalid data
  - "strict" to raise an UnicodeDecodeError exception for invalid data

**Raises**

- DXFStructureError – for invalid or corrupted DXF structures
- UnicodeDecodeError – if errors is “strict” and a decoding error occurs

```python
ezdxf.recover.explore(filename: str, errors: str = 'ignore') \rightarrow Tuple[Drawing, Auditor]
```

Read a DXF document from file system similar to `readfile()`, but this function will use a special tag loader, which synchronise the tag stream if invalid tags occur. This function is intended to load corrupted DXF files and should only be used to explore such files, data loss is very likely.

**Parameters**

- **filename** – file-system name of the DXF document to load
- **errors** – specify decoding error handler
  - "surrogateescape” to preserve possible binary data (default)
  - "ignore” to use the replacement char U+FFFD “” for invalid data
  - "strict” to raise an UnicodeDecodeError exception for invalid data

**Raises**
6.8.2 DXF Structures

Sections

Header Section

The drawing settings are stored in the HEADER section, which is accessible by the header attribute of the Drawing object. See the online documentation from Autodesk for available header variables.

See also:
DXF Internals: HEADER Section

class ezdxf.sections.header.HeaderSection

- **custom_vars**
  Stores the custom drawing properties in a CustomVars object.

- **len** () → int
  Returns count of header variables.

- **contains**(key) → bool
  Returns True if header variable key exist.

- **varnames** () → KeysView[KT]
  Returns an iterable of all header variable names.

- **get**(key: str, default: Any = None) → Any
  Returns value of header variable key if exist, else the default value.

- **getitem**(key: str) → Any
  Get header variable key by index operator like: drawing.header['$ACADVER']

- **setitem**(key: str, value: Any) → None
  Set header variable key to value by index operator like: drawing.header['$ANGDIR'] = 1

- **delitem**(key: str) → None
  Delete header variable key by index operator like: del drawing.header['$ANGDIR']

class ezdxf.sections.header.CustomVars

Stores custom properties in the DXF header as $CUSTOMPROPERTYTAG and $CUSTOMPROPERTY values. Custom properties are just supported by DXF R2004 (AC1018) or later. ezdxf can create custom properties at older DXF versions, but AutoCAD will not show this properties.

- **properties**
  List of custom drawing properties, stored as string tuples (tag, value). Multiple occurrence of the same custom tag is allowed, but not well supported by the interface. This is a standard python list and it is save to change this list as long you store just tuples of strings in the format (tag, value).

- **len** () → int
  Count of custom properties.

- **iter** () → Iterable[Tuple[str, str]]
  Iterate over all custom properties as (tag, value) tuples.
clear() → None
Remove all custom properties.

get (tag: str, default: str = None)
Returns the value of the first custom property tag.

has_tag (tag: str) → bool
Returns True if custom property tag exist.

append (tag: str, value: str) → None
Add custom property as (tag, value) tuple.

replace (tag: str, value: str) → None
Replaces the value of the first custom property tag by a new value.

remove (tag: str, all: bool = False) → None
Removes the first occurrence of custom property tag, removes all occurrences if all is True.

Classes Section

The CLASSES section in DXF files holds the information for application-defined classes whose instances appear in Layout objects. As usual package user there is no need to bother about CLASSES.

See also:

DXF Internals: CLASSES Section

class ezdxf.sections.classes.ClassesSection

classes
Storage of all DXFClass objects, they are not stored in the entities database, because CLASS has no handle attribute.

register (classes: Iterable[DXFClass])

add_class (name: str)
Register a known class by name.

get (name: str) → DXFClass
Returns the first class matching name.

Storage key is the (name, cpp_class_name) tuple, because there are some classes with the same name but different cpp_class_names.

add_required_classes (name: str) → DXFClass
Add all required CLASS definitions for dxfversion.

update_instance_counters () → None
Update CLASS instance counter for all registered classes, requires DXF R2004+.

class ezdxf.entities.DXFClass
Information about application-defined classes.

dxf.name
Class DXF record name.

dxf.cpp_class_name
C++ class name. Used to bind with software that defines object class behavior.
**App Name**
Application name. Posted in Alert box when a class definition listed in this section is not currently loaded.

**Flags**
Proxy capabilities flag

<table>
<thead>
<tr>
<th>Flag Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No operations allowed (0)</td>
</tr>
<tr>
<td>1</td>
<td>Erase allowed (0x1)</td>
</tr>
<tr>
<td>2</td>
<td>Transform allowed (0x2)</td>
</tr>
<tr>
<td>4</td>
<td>Color change allowed (0x4)</td>
</tr>
<tr>
<td>8</td>
<td>Layer change allowed (0x8)</td>
</tr>
<tr>
<td>16</td>
<td>Linetype change allowed (0x10)</td>
</tr>
<tr>
<td>32</td>
<td>Linetype scale change allowed (0x20)</td>
</tr>
<tr>
<td>64</td>
<td>Visibility change allowed (0x40)</td>
</tr>
<tr>
<td>128</td>
<td>Cloning allowed (0x80)</td>
</tr>
<tr>
<td>256</td>
<td>Lineweight change allowed (0x100)</td>
</tr>
<tr>
<td>512</td>
<td>Plot Style Name change allowed (0x200)</td>
</tr>
<tr>
<td>895</td>
<td>All operations except cloning allowed (0x37F)</td>
</tr>
<tr>
<td>1023</td>
<td>All operations allowed (0x3FF)</td>
</tr>
<tr>
<td>1024</td>
<td>Disables proxy warning dialog (0x400)</td>
</tr>
<tr>
<td>32768</td>
<td>R13 format proxy (0x8000)</td>
</tr>
</tbody>
</table>

**Instance Count**
Instance count for a custom class.

**Was a Proxy**
Set to 1 if class was not loaded when this DXF file was created, and 0 otherwise.

**Is an Entity**
Set to 1 if class was derived from the *DXFGraphic* class and can reside in layouts. If 0, instances may appear only in the OBJECTS section.

**Key**
Unique name as (*name*, *cpp_class_name*) tuple.

### Tables Section

The TABLES section is the home of all TABLE objects of a DXF document.

See also:
DXF Internals: TABLES Section

class ezdxf.sections.tables.TablesSection

    layers
    LayerTable object for Layer objects

    linetypes
    Generic Table object for Linetype objects

    styles
    StyleTable object forTextStyle objects

    dimstyles
    Generic Table object for DimStyle objects
appids
   Generic Table object for AppID objects

ucs
   Generic Table object for UCSTable objects

views
   Generic Table object for View objects

viewports
   ViewportTable object for VPort objects

block_records
   Generic Table object for BlockRecord objects

Blocks Section

The BLOCKS section is the home all block definitions (BlockLayout) of a DXF document.

See also:
DXF Internals: BLOCKS Section and Block Management Structures

class ezdxf.sections.blocks.BlocksSection

   __iter__( ) → Iterable[BlockLayout]
      Iterable of all BlockLayout objects.

   __contains__( name: str ) → bool
      Returns True if BlockLayout name exist.

   __getitem__( name: str ) → BlockLayout
      Returns BlockLayout name, raises DXFKeyError if name not exist.

   __delitem__( name: str ) → None
      Deletes BlockLayout name and all of its content, raises DXFKeyError if name not exist.

   get( self, name: str, default=None ) → BlockLayout
      Returns BlockLayout name, returns default if name not exist.

   new( name: str, base_point: Sequence[float] = (0, 0), dxfattribs: dict = None ) → BlockLayout
      Create and add a new BlockLayout, name is the BLOCK name, base_point is the insertion point of the BLOCK.

   new_anonymous_block( type_char: str = 'U', base_point: Sequence[float] = (0, 0) ) → BlockLayout
      Create and add a new anonymous BlockLayout, type_char is the BLOCK type, base_point is the insertion point of the BLOCK.

   rename_block( old_name: str, new_name: str ) → None
      Rename BlockLayout old_name to new_name

   type_char | Anonymous Block Type
   +----------+----------------------------------------
   'U'       | '*U###' anonymous BLOCK
   'E'       | '*E###' anonymous non-uniformly scaled BLOCK
   'X'       | '*X###' anonymous HATCH graphic
   'D'       | '*D###' anonymous DIMENSION graphic
   'A'       | '*A###' anonymous GROUP
   'T'       | '*T###' anonymous block for ACAD_TABLE content

6.8. Reference
**delete_block** *(name: str, safe: bool = True) → None*

Delete block. If *safe* is True, check if block is still referenced.

**Parameters**

- `name` – block name (case insensitive)
- `safe` – check if block is still referenced or special block without explicit references

**Raises**

- `DXFKeyError` – if block not exists
- `DXFBlockInUseError` – if block is still referenced, and save is True

**delete_all_blocks** *(*)

Delete all blocks without references except modelspace- or paperspace layout blocks, special arrow- and anonymous blocks (DIMENSION, ACAD_TABLE).

**Warning:** There could exist undiscovered references to blocks which are not documented in the DXF reference, hidden in extended data sections or application defined data, which could produce invalid DXF documents if such referenced blocks will be deleted.

Changed in version 0.14: removed unsafe mode

**Entities Section**

The ENTITIES section is the home of all *Modelspace* and active *Paperspace* layout entities. This is a real section in the DXF file, for *ezdxf* the *EntitySection* is just a proxy for modelspace and the active paperspace linked together.

**See also:**

DXF Internals: *ENTITIES Section*

```python
class ezdxf.sections.entities.EntitySection

__iter__() → Iterable[DXFEntity]
    Iterable for all entities of modelspace and active paperspace.

__len__() → int
    Returns count of all entities of modelspace and active paperspace.
```

**Objects Section**

The OBJECTS section is the home of all none graphical objects of a DXF document. The OBJECTS section is accessible by *Drawing.objects*.

Convenience methods of the *Drawing* object to create required structures in the OBJECTS section:

- **IMAGEDEF**: `add_image_def()`
- **UNDERLAYDEF**: `add_underlay_def()`
- **RASTERVARIABLES**: `set_raster_variables()`
- **WIPEOUTVARIABLES**: `set_wipeout_variables()`
See also:

DXF Internals: **OBJECTS Section**

```python
class ezdxf.sections.objects.ObjectsSection
```

**rootdict**

Root dictionary.

```python
__len__() → int
```

Returns count of DXF objects.

```python
__iter__() → Iterable[DXFObject]
```

Returns iterable of all DXF objects in the OBJECTS section.

```python
__getitem__(index) → DXFObject
```

Get entity at `index`

The underlying data structure for storing DXF objects is organized like a standard Python list, therefore `index` can be any valid list indexing or slicing term, like a single index `objects[-1]` to get the last entity, or an index slice `objects[:10]` to get the first 10 or less objects as `List[DXFObject]`.

```python
__contains__(entity: Union[DXFObject, str]) → bool
```

Returns `True` if `entity` stored in OBJECTS section.

### Parameters

- **entity** — DXFObject or handle as hex string

```python
query(query: str = '*') → ezdxf.query.EntityQuery
```

Get all DXF objects matching the **Entity Query String**.

```python
add_dictionary(owner: str = '0', hard_owned: bool = False) →
```

Add new `Dictionary` object.

### Parameters

- **owner** — handle to owner as hex string.
- **hard_owned** — `True` to treat entries as hard owned.

```python
add_dictionary_with_default(owner='0', default='0', hard_owned: bool = False) →
```

Add new `DictionaryWithDefault` object.

### Parameters

- **owner** — handle to owner as hex string.
- **default** — handle to default entry.
- **hard_owned** — `True` to treat entries as hard owned.

```python
add_dictionary_var(owner: str = '0', value: str = '') →
```

Add a new `DictionaryVar` object.

### Parameters

- **owner** — handle to owner as hex string.
- **value** — value as string

```python
add_geodata(owner: str = '0', dxfattribs: dict = None) → GeoData
```

Creates a new GeoData entity and replaces existing ones. The GEODATA entity resides in the OBJECTS section and NOT in the layout entity space and it is linked to the layout by an extension dictionary located in BLOCK_RECORD of the layout.

---

6.8. Reference
The GEODATA entity requires DXF version R2010+. The DXF Reference does not document if other layouts than model space supports geo referencing, so getting/setting geo data may only make sense for the model space layout, but it is also available in paper space layouts.

**Parameters**

- **owner** – handle to owner as hex string
- **dxfattribs** – DXF attributes for GeoData entity

**add_image_def** *(filename: str, size_in_pixel: Tuple[int, int], name=None) → ImageDef*

Add an image definition to the objects section.

Add an ImageDef entity to the drawing (objects section). *filename* is the image file name as relative or absolute path and *size_in_pixel* is the image size in pixel as (x, y) tuple. To avoid dependencies to external packages, ezdxf can not determine the image size by itself. Returns a ImageDef entity which is needed to create an image reference. *name* is the internal image name, if set to None, name is auto-generated.

Absolute image paths works best for AutoCAD but not really good, you have to update external references manually in AutoCAD, which is not possible in TrueView. If the drawing units differ from 1 meter, you also have to use: `set_raster_variables()`.

**Parameters**

- **filename** – image file name (absolute path works best for AutoCAD)
- **size_in_pixel** – image size in pixel as (x, y) tuple
- **name** – image name for internal use, None for using filename as name (best for AutoCAD)

**add_placeholder** *(owner: str = '0') → Placeholder*

Add a new Placeholder object.

**Parameters**

- **owner** – handle to owner as hex string.

**add_underlay_def** *(filename: str, format: str = 'pdf', name: str = None) → UnderlayDef*

Add an UnderlayDef entity to the drawing (OBJECTS section). *filename* is the underlay file name as relative or absolute path and *format* as string (pdf, dwf, dgn). The underlay definition is required to create an underlay reference.

**Parameters**

- **filename** – underlay file name
- **format** – file format as string 'pdf' | 'dwf' | 'dgn' or 'ext' for getting file format from filename extension
- **name** – pdf format = page number to display; dgn format = 'default'; dwf: ????

**add_xrecord** *(owner: str = '0') → XRecord*

Add a new XRecord object.

**Parameters**

- **owner** – handle to owner as hex string.

**set_raster_variables** *(frame: int = 0, quality: int = 1, units: str = 'm') → None*

Set raster variables.

**Parameters**

- **frame** – 0 = do not show image frame; 1 = show image frame
- **quality** – 0 = draft; 1 = high
- **units** – units for inserting images. This defines the real world unit for one drawing unit for the purpose of inserting and scaling images with an associated resolution.
(internal API), public interface `set_raster_variables()`

```python
set_wipeout_variables(frame: int = 0) → None
```

Set wipeout variables.

**Parameters**

- `frame` –
  - `0` = do not show image frame; `1` = show image frame

Tables

Table Classes

Generic Table Class

```python
class ezdxf.sections.table.Table
```

Generic collection of table entries. Table entry names are case insensitive: `'Test' == 'TEST'`.  

```python
static key(entity: Union[str, DXFEntity]) → str
```

Unified table entry key.

```python
has_entry(name: Union[str, DXFEntity]) → bool
```

Returns `True` if an table entry `name` exist.

```python
__contains__(name: Union[str, DXFEntity]) → bool
```

Returns `True` if an table entry `name` exist.

```python
__len__() → int
```

Count of table entries.

```python
__iter__() → Iterable[DXFEntity]
```

Iterable of all table entries.

```python
new(name: str, dxfattribs: dict = None) → DXFEntity
```

Create a new table entry `name`.

**Parameters**

- `name` – name of table entry, case insensitive
- `dxfattribs` – additional DXF attributes for table entry

```python
get(name: str) → DXFEntity
```

Get table entry `name` (case insensitive). Raises `DXFValueError` if table entry does not exist.

```python
remove(name: str) → None
```

Removes table entry `name`. Raises `DXFValueError` if table entry does not exist.

```python
duplicate_entry(name: str, new_name: str) → DXFEntity
```

Returns a new table entry `new_name` as copy of `name`, replaces entry `new_name` if already exist.
Raises DXFValueError – name does not exist

Layer Table

class ezdxf.sections.table.LayerTable
    Subclass of Table.
    Collection of Layer objects.

Linetype Table

Generic table class of Table.
Collection of Linetype objects.

Style Table

class ezdxf.sections.table.StyleTable
    Subclass of Table.
    Collection of Textstyle objects.

    get_shx(shxname: str) → Textstyle
        Get existing shx entry, or create a new entry.

    Parameters shxname – shape file name like ‘ltypeshp.lin’

    find_shx(shxname: str) → Optional[Textstyle]
        Find .shx shape file table entry, by a case insensitive search.
        A .shx shape file table entry has no name, so you have to search by the font attribute.

        Parameters shxname – .shx shape file name

DimStyle Table

Generic table class of Table.
Collection of DimStyle objects.

AppID Table

Generic table class of Table.
Collection of AppID objects.

UCS Table

Generic table class of Table.
Collection of UCSTable objects.
**View Table**

Generic table class of `Table`. Collection of `View` objects.

**Viewport Table**

class ezdxfs.sections.table.ViewportTable

The viewport table stores the modelspace viewport configurations. A viewport configuration is a tiled view of multiple viewports or just one viewport. In contrast to other tables the viewport table can have multiple entries with the same name, because all viewport entries of a multi-viewport configuration are having the same name - the viewport configuration name.

The name of the actual displayed viewport configuration is 'ACTIVE'.

Duplication of table entries is not supported: `duplicate_entry()` raises `NotImplementedError`

`get_config`(self, name: str) → List[Viewport]

Returns a list of `Viewport` objects, for the multi-viewport configuration `name`.

`delete_config`(name: str) → None

Delete all `Viewport` objects of the multi-viewport configuration `name`.

**Block Record Table**

Generic table class of `Table`. Collection of `BlockRecord` objects.

**Layer**

LAYER (DXF Reference) definition, defines attribute values for entities on this layer for their attributes set to BYLAYER.

Subclass of | `ezdxfs.entities.DXFEntity`
---|---
DXF type | 'LAYER'
Factory function | `Drawing.layers.new()`

See also:

*Layer Concept* and *Tutorial for Layers*

class ezdxfs.entities.Layer

```
    dxf.handle
    DXF handle (feature for experts)

    dxf.owner
    Handle to owner (LayerTable).

    dxf.name
    Layer name, case insensitive and can not contain any of this characters: <>/":;:?*|= (str)
```

6.8. Reference
**dxflflags**
Layer flags (bit-coded values, feature for experts)

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Layer is frozen; otherwise layer is thawed; use <code>is_frozen().freeze()</code> and <code>thaw()</code></td>
</tr>
<tr>
<td>2</td>
<td>Layer is frozen by default in new viewports</td>
</tr>
<tr>
<td>4</td>
<td>Layer is locked; use <code>is_locked().lock().unlock()</code></td>
</tr>
<tr>
<td>16</td>
<td>If set, table entry is externally dependent on an xref</td>
</tr>
<tr>
<td>32</td>
<td>If both this bit and bit 16 are set, the externally dependent xref has been successfully resolved</td>
</tr>
<tr>
<td>64</td>
<td>If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is for the benefit of AutoCAD commands. It can be ignored by most programs that read DXF files and need not be set by programs that write DXF files)</td>
</tr>
</tbody>
</table>

**dxfcolor**
Layer color, but use property `Layer.color` to get/set color value, because color is negative for layer status `off` (int)

**dxfture_color**
Layer true color value as int, use property `Layer.rgb` to set/get true color value as `(r, g, b)` tuple. (requires DXF R2004)

**dxflinetype**
Name of line type (str)

**dxf.plot**
Plot flag (int). Whether entities belonging to this layer should be drawn when the document is exported (plotted) to pdf. Does not affect visibility inside the CAD application itself.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>plot layer (default value)</td>
</tr>
<tr>
<td>0</td>
<td>don’t plot layer</td>
</tr>
</tbody>
</table>

**dxflineweight**
Line weight in mm times 100 (e.g. 0.13mm = 13). Smallest line weight is 13 and biggest line weight is 200, values outside this range prevents AutoCAD from loading the file. 

`ezdxf.lldxf.const.LINEWEIGHT_DEFAULT` for using global default line weight. (requires DXF R13)

**dxf.plotstyle_handle**
Handle to plot style name? (requires DXF R13)

**dxf.material_handle**
Handle to default `Material`. (requires DXF R13)

**rgb**
Get/set DXF attribute `dxf.true_color` as `(r, g, b)` tuple, returns `None` if attribute `dxf.true_color` is not set.

```python
layer.rgb = (30, 40, 50)
r, g, b = layer.rgb
```

This is the recommend method to get/set RGB values, when ever possible do not use the DXF low level attribute `dxf.true_color`. 
color
Get/set layer color, preferred method for getting the layer color, because \texttt{dxf.color} is negative for layer status \textit{off}.

description
Get/set layer description as string

transparency
Get/set layer transparency as float value in the range from 0 to 1. 0 for no transparency (opaque) and 1 for 100% transparency.

\texttt{is\_frozen()} → bool
Returns \textbf{True} if layer is frozen.

\texttt{freeze()} → None
Freeze layer.

\texttt{thaw()} → None
Thaw layer.

\texttt{is\_locked()} → bool
Returns \textbf{True} if layer is locked.

\texttt{lock()} → None
Lock layer, entities on this layer are not editable - just important in CAD applications.

\texttt{unlock()} → None
Unlock layer, entities on this layer are editable - just important in CAD applications.

\texttt{is\_off()} → bool
Returns \textbf{True} if layer is off.

\texttt{is\_on()} → bool
Returns \textbf{True} if layer is on.

\texttt{on()} → None
Switch layer \textit{on} (visible).

\texttt{off()} → None
Switch layer \textit{off} (invisible).

\texttt{get\_color()} → int
Use property \texttt{Layer.color} instead.

\texttt{set\_color(value: int)} → None
Use property \texttt{Layer.color} instead.

\texttt{rename(name: str)} → None
Rename layer and all known (documented) references to this layer.

\begin{footnotesize}
\begin{itemize}
\item \textbf{ValueError} – \textit{name} contains invalid characters: \/":;?*|='
\item \textbf{ValueError} – layer \textit{name} already exist
\item \textbf{ValueError} – renaming of layers '0' and 'DEFPOINTS' not possible
\end{itemize}
\end{footnotesize}
**Style**

**Important:** DXF is not a layout preserving data format like PDF. It is more similar to the MS Word format. Many applications can open MS Word documents, but the displayed or printed document does not look perfect like the result of MS Word.

The final rendering of DXF files is highly dependent on the interpretation of DXF entities by the rendering engine, and the DXF reference does not provide any guidelines for rendering entities. The biggest visual differences of CAD applications are the text renderings, therefore the only way to get the exact same result is to use the same CAD application.

The DXF format does not and **can not** embed TTF fonts like the PDF format!

The **Textstyle** entity defines a text style (DXF Reference), and can be used by the entities: **Text, Attrib, Attdef, MText, Dimension, Leader** and **MultiLeader**.

Example to create a new text style “Arial” and to apply this text style:

```python
doc.styles.new("Arial", dfxattrs=\{"font": "Arial.ttf"\})
msp = doc.modelspace()
msp.add_text("my text", dfxattrs=\{"style": "Arial"\})
```

The settings stored in the **Textstyle** entity are the default text style values used by CAD applications if the text settings are not stored in the text entity itself. But not all setting are substituted by the default value. The **height** or **width** attribute must be stored in the text entities itself in order to influence the appearance of the text. It is **recommended** that you do not rely on the default settings in the **Textstyle** entity, set all attributes in the text entity itself if supported.

**Font Settings**

Just a few settings are available exclusive by the **Textstyle** entity:

The most important setting is the **font** attribute, this attribute defines the rendering font as raw TTF file name, e.g. “Arial.ttf” or “OpenSansCondensed-Light.ttf”, this file name is often not the name displayed in GUI application and you have to digg down into the fonts folder e.g. (C:\Windows\Fonts) to get the the real file name for the TTF font. Do not include the path!
AutoCAD supports beyond the legacy SHX fonts only TTF fonts. The SHX font format is not documented and only available in some CAD applications. The `ezdxf drawing` add-on replaces the SHX fonts by TTF fonts, which look similar to the SHX fonts, unfortunately the license of these fonts is unclear, therefore they can not be packaged with `ezdxf`. They are installed automatically if you use an Autodesk product like TrueView, or search the internet at you own risk for these TTF fonts.

The extended font data can provide extra information for the font, it is stored in the XDATA section, not well documented and not widely supported.

**Important:** The DXF format does not and **can not** embed TTF fonts like the PDF format!

You need to make sure that the CAD application is properly configured to have access to the system fonts. The DXF format has no setting where the CAD application should search for fonts, and does not guarantee that the text rendering on other computers or operating systems looks the same as on your current system on which you created the DXF.

The second exclusive setting is the vertical text flag in `Textstyle.flags`. The vertical text style is enabled for **all** entities using the text style. Vertical text works only for SHX fonts and is not supported for TTF fonts (in AutoCAD) and is works only for the single line entities `Text` and `Attrib`. Most CAD applications beside AutoCAD and BricsCAD do not support vertical text rendering and even AutoCAD and BricsCAD have problems with vertical text rendering in some circumstances. Using the vertical text feature is not recommended.
Subclass of  \texttt{ezdxf.entities.DXFEntity}
\begin{tabular}{|l|l|}
\hline
\textbf{DXF type} & 'STYLE' \\
\hline
\textbf{Factory function} & \texttt{Drawing.styles.new()} \\
\hline
\end{tabular}

\textbf{See also:}

\emph{Tutorial for Text} and DXF internals for \emph{DIMSTYLE Table}.

class \texttt{ezdxf.entities.Textstyle}

\begin{verbatim}
dxf.handle
    DXF handle (feature for experts).
dxf.owner
    Handle to owner (StyleTable).
dxf.name
    Style name (str)
dxf.flags
    Style flags (feature for experts).
\end{verbatim}

\begin{tabular}{|c|p{0.8\textwidth}|}
\hline
1 & If set, this entry describes a shape \\
4 & Vertical text \\
16 & If set, table entry is externally dependent on an xref \\
32 & If both this bit and bit 16 are set, the externally dependent xref has been successfully resolved \\
64 & If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is only for the benefit of AutoCAD commands. It can be ignored by most programs that read DXF files and need not be set by programs that write DXF files) \\
\hline
\end{tabular}

\begin{verbatim}
dxf.height
    Fixed height in drawing units as float value, 0 for not fixed.
dxf.width
    Width factor as float value, default value is 1.
dxf.oblique
    Oblique (slanting) angle in degrees as float value, default value is 0 for no slanting.
dxf.generation_flags
    Text generations flags as int value.
\end{verbatim}

\begin{tabular}{|c|p{0.8\textwidth}|}
\hline
2 & text is backward (mirrored along the x-axis) \\
4 & text is upside down (mirrored about the base line) \\
\hline
\end{tabular}

\begin{verbatim}
dxf.last_height
    Last height used in drawing units as float value.
dxf.font
    Raw font file name as string without leading path, e.g. “Arial.ttf” for TTF fonts or the SHX font name like “TXT” or “TXT.SHX”.
dxf.bigfont
    Big font name as string, blank if none. No documentation how to use this feature, maybe just a legacy artifact.
\end{verbatim}
get_extended_font_data() \rightarrow \text{Tuple[str, bool, bool]}
Returns extended font data as tuple (font-family, italic-flag, bold-flag).

The extended font data is optional and not reliable! Returns ("", False, False) if extended font data is not present.

set_extended_font_data(family: str = ", *, italic=False, bold=False) \rightarrow \text{None}
Set extended font data, the font-family name family is not validated by ezdxf. Overwrites existing data.

discard_extended_font_data()
Discard extended font data.

Linetype
Defines a linetype (DXF Reference).

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFEntity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'LTYPE'</td>
</tr>
<tr>
<td>Factory function</td>
<td>Drawing.linetypes.new()</td>
</tr>
</tbody>
</table>

See also:

Tutorial for Linetypes

DXF Internals: LTYPE Table

class ezdxf.entities.Linetype

dxf.name
Linetype name (str).

dxf.owner
Handle to owner (Table).

dxf.description
Linetype description (str).

dxf.length
Total pattern length in drawing units (float).

dxf.items
Number of linetype elements (int).

DimStyle

DIMSTYLE (DXF Reference) defines the appearance of Dimension entities. Each of this dimension variables starting with dim... can be overridden for any Dimension entity individually.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFEntity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'DIMSTYLE'</td>
</tr>
<tr>
<td>Factory function</td>
<td>Drawing.dimstyles.new()</td>
</tr>
</tbody>
</table>
class ezdxf.entities.DimStyle

    dxf.owner
        Handle to owner (Table).

    dxf.name
        Dimension style name.

    dxf.flags
        Standard flag values (bit-coded values):

        | Flag | Description |
        |------|-------------|
        | 16   | If set, table entry is externally dependent on an xref |
        | 32   | If both this bit and bit 16 are set, the externally dependent xref has been successfully resolved |
        | 64   | If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is only for the benefit of AutoCAD) |

    dxf.dimpost
        Prefix/suffix for primary units dimension values.

    dxf.dimapost
        Prefix/suffix for alternate units dimensions.

    dxf.dimblk
        Block type to use for both arrowheads as name string.

    dxf.dimblk1
        Block type to use for first arrowhead as name string.

    dxf.dimblk2
        Block type to use for second arrowhead as name string.

    dxf.dimscale
        Global dimension feature scale factor. (default=1)

    dxf.dimasz
        Dimension line and arrowhead size. (default=0.25)

    dxf.dimexo
        Distance from origin points to extension lines. (default imperial=0.0625, default metric=0.625)

    dxf.dimdli
        Incremental spacing between baseline dimensions. (default imperial=0.38, default metric=3.75)

    dxf.dimexe
        Extension line distance beyond dimension line. (default imperial=0.28, default metric=2.25)

    dxf.dimrnd
        Rounding value for decimal dimensions. (default=0)

        Rounds all dimensioning distances to the specified value, for instance, if DIMRND is set to 0.25, all distances round to the nearest 0.25 unit. If you set DIMRND to 1.0, all distances round to the nearest integer.

    dxf.dimdle
        Dimension line extension beyond extension lines. (default=0)

    dxf.dimtp
        Upper tolerance value for tolerance dimensions. (default=0)

    dxf.dimtm
        Lower tolerance value for tolerance dimensions. (default=0)
**ezdxf Documentation, Release 0.16.2**

- **dxfldimtxt**
  
  Size of dimension text. (default imperial=0.28, default metric=2.5)

- **dxfldimcen**
  
  Controls placement of center marks or centerlines. (default imperial=0.09, default metric=2.5)

- **dxfldimtksz**
  
  Controls size of dimension line tick marks drawn instead of arrowheads. (default=0)

- **dxfldimalf**
  
  Alternate units dimension scale factor. (default=25.4)

- **dxfldimlfac**
  
  Scale factor for linear dimension values. (default=1)

- **dxfldimtvp**
  
  Vertical position of text above or below dimension line if `dimtad` is 0. (default=0)

- **dxfldimtfac**
  
  Scale factor for fractional or tolerance text size. (default=1)

- **dxfldimgap**
  
  Gap size between dimension line and dimension text. (default imperial=0.09, default metric=0.625)

- **dxfldimltdnrd**
  
  Rounding value for alternate dimension units. (default=0)

- **dxfldimtol**
  
  Toggles creation of appended tolerance dimensions. (default imperial=1, default metric=0)

- **dxfldimlim**
  
  Toggles creation of limits-style dimension text. (default=0)

- **dxfldimth**
  
  Orientation of text inside extension lines. (default imperial=1, default metric=0)

- **dxfldimtoh**
  
  Orientation of text outside extension lines. (default imperial=1, default metric=0)

- **dxfldimsel**
  
  Toggles suppression of first extension line. (default=0)

- **dxfldimese2**
  
  Toggles suppression of second extension line. (default=0)

- **dxfldimtad**
  
  Sets vertical text placement relative to dimension line. (default imperial=0, default metric=1)

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>center</td>
</tr>
<tr>
<td>1</td>
<td>above</td>
</tr>
<tr>
<td>2</td>
<td>outside, handled like above by <code>ezdxf</code></td>
</tr>
<tr>
<td>3</td>
<td>JIS, handled like above by <code>ezdxf</code></td>
</tr>
<tr>
<td>4</td>
<td>below</td>
</tr>
</tbody>
</table>

- **dxfldimzin**
  
  Zero suppression for primary units dimensions. (default imperial=0, default metric=8)

  Values 0-3 affect feet-and-inch dimensions only.
## ezdxf Documentation, Release 0.16.2

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Suppresses zero feet and precisely zero inches</td>
</tr>
<tr>
<td>1</td>
<td>Includes zero feet and precisely zero inches</td>
</tr>
<tr>
<td>2</td>
<td>Includes zero feet and suppresses zero inches</td>
</tr>
<tr>
<td>3</td>
<td>Includes zero inches and suppresses zero feet</td>
</tr>
<tr>
<td>4</td>
<td>Suppresses leading zeros in decimal dimensions (for example, 0.5000 becomes .5000)</td>
</tr>
<tr>
<td>8</td>
<td>Suppresses trailing zeros in decimal dimensions (for example, 12.5000 becomes 12.5)</td>
</tr>
<tr>
<td>12</td>
<td>Suppresses both leading and trailing zeros (for example, 0.5000 becomes .5)</td>
</tr>
</tbody>
</table>

**dxf.dimazin**

Controls zero suppression for angular dimensions. (default=0)

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Displays all leading and trailing zeros</td>
</tr>
<tr>
<td>1</td>
<td>Suppresses leading zeros in decimal dimensions (for example, 0.5000 becomes .5000)</td>
</tr>
<tr>
<td>2</td>
<td>Suppresses trailing zeros in decimal dimensions (for example, 12.5000 becomes 12.5)</td>
</tr>
<tr>
<td>3</td>
<td>Suppresses leading and trailing zeros (for example, 0.5000 becomes .5)</td>
</tr>
</tbody>
</table>

**dxf.dimalt**

Enables or disables alternate units dimensioning. (default=0)

**dxf.dimaltd**

Controls decimal places for alternate units dimensions. (default imperial=2, default metric=3)

**dxf.dimtofl**

Toggles forced dimension line creation. (default imperial=0, default metric=1)

**dxf.dimsah**

Toggles appearance of arrowhead blocks. (default=0)

**dxf.dimtix**

Toggles forced placement of text between extension lines. (default=0)

**dxf.dimsoxd**

Suppresses dimension lines outside extension lines. (default=0)

**dxf.dimclrd**

Dimension line, arrowhead, and leader line color. (default=0)

**dxf.dimclre**

Dimension extension line color. (default=0)

**dxf.dimclrt**

Dimension text color. (default=0)

**dxf.dimadec**

Controls the number of decimal places for angular dimensions.

**dxf.dimunit**

Obsolete, now use DIMLUNIT AND DIMFRAC

**dxf.dimdec**

Decimal places for dimension values. (default imperial=4, default metric=2)

**dxf.dimtdec**

Decimal places for primary units tolerance values. (default imperial=4, default metric=2)

**dxf.dimaltu**

Units format for alternate units dimensions. (default=2)
dxp.dimaltttd
Decimal places for alternate units tolerance values. (default imperial=4, default metric=2)

dxp.dimaunit
Unit format for angular dimension values. (default=0)

dxp.dimfrac
Controls the fraction format used for architectural and fractional dimensions. (default=0)

dxp.dimlunit
Specifies units for all nonangular dimensions. (default=2)

dxp.dimdsep
Specifies a single character to use as a decimal separator. (default imperial = ' ', default metric = ', ')
This is an integer value, use ord(' ') to write value.

dxp.dimtmove
Controls the format of dimension text when it is moved. (default=0)

<table>
<thead>
<tr>
<th></th>
<th>Moves the dimension line with dimension text</th>
<th>Adds a leader when dimension text is moved</th>
<th>Allows text to be moved freely without a leader</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dxp.dimjust
Horizontal justification of dimension text. (default=0)

<table>
<thead>
<tr>
<th></th>
<th>Center of dimension line</th>
<th>Left side of the dimension line, near first extension line</th>
<th>Right side of the dimension line, near second extension line</th>
<th>Over first extension line</th>
<th>Over second extension line</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dxp.dimsd1
Toggles suppression of first dimension line. (default=0)

dxp.dimsd2
Toggles suppression of second dimension line. (default=0)

dxp.dimtolj
Vertical justification for dimension tolerance text. (default=1)

<table>
<thead>
<tr>
<th></th>
<th>Align with bottom line of dimension text</th>
<th>Align vertical centered to dimension text</th>
<th>Align with top line of dimension text</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dxp.dimtizin
Zero suppression for tolerances values, see DimStyle.dxf.dimzin

dxp.dimaltz
Zero suppression for alternate units dimension values. (default=0)

dxp.dimalttz
Zero suppression for alternate units tolerance values. (default=0)

dxp.dimfit
Obsolete, now use DIMATFIT and DIMTMOVE
ezdxf Documentation, Release 0.16.2

```python
dxf.dimupt
  Controls user placement of dimension line and text. (default=0)

dxf.dimatfit
  Controls placement of text and arrowheads when there is insufficient space between the extension lines. (default=3)

dxf.dimtxsty
  Text style used for dimension text by name.

dxf.dimtxsty_handle
  Text style used for dimension text by handle of STYLE entry. (use DimStyle.dxf.dimtxsty to get/set text style by name)

dxf.dimldrblk
  Specify arrowhead used for leaders by name.

dxf.dimldrblk_handle
  Specify arrowhead used for leaders by handle of referenced block. (use DimStyle.dfx.dimldrblk to get/set arrowhead by name)

dxf.dimbblk_handle
  Block type to use for both arrowheads, handle of referenced block. (use DimStyle.dfx.dimbblk to get/set arrowheads by name)

dxf.dimbblk1_handle
  Block type to use for first arrowhead, handle of referenced block. (use DimStyle.dfx.dimbblk1 to get/set arrowhead by name)

dxf.dimbblk2_handle
  Block type to use for second arrowhead, handle of referenced block. (use DimStyle.dfx.dimbblk2 to get/set arrowhead by name)

dxf.dimlwd
  Lineweight value for dimension lines. (default=-2, BYBLOCK)

dxf.dimlwe
  Lineweight value for extension lines. (default=-2, BYBLOCK)

dxf.dimtype
  Specifies the linetype used for the dimension line as linetype name, requires DXF R2007+

dxf.dimtype_handle
  Specifies the linetype used for the dimension line as handle to LTYPE entry, requires DXF R2007+ (use DimStyle.dfx.dimtype to get/set linetype by name)

dxf.dimltex1
  Specifies the linetype used for the extension line 1 as linetype name, requires DXF R2007+

dxf.dimlex1_handle
  Specifies the linetype used for the extension line 1 as handle to LTYPE entry, requires DXF R2007+ (use DimStyle.dfx.dimlex1 to get/set linetype by name)

dxf.dimltex2
  Specifies the linetype used for the extension line 2 as linetype name, requires DXF R2007+

dxf.dimlex2_handle
  Specifies the linetype used for the extension line 2 as handle to LTYPE entry, requires DXF R2007+ (use DimStyle.dfx.dimlex2 to get/set linetype by name)

dxf.dimfxlon
  Extension line has fixed length if set to 1, requires DXF R2007+
```
ezdxf Documentation, Release 0.16.2

**dx.dfxl**
Length of extension line below dimension line if fixed (DimStyle.dxf.dimtfxlon == 1), DimStyle.dxf.dimexen defines the the length above the dimension line, requires DXF R2007+

**dx.dfmtfill**
Text fill 0=off; 1=background color; 2=custom color (see DimStyle.dxf.dfmtfillclr), requires DXF R2007+

**dx.dfmtfillclr**
Text fill custom color as color index (1-255), requires DXF R2007+

**copy_to_header** (dwg: Drawing) → None
Copy all dimension style variables to HEADER section of doc.

**set_arrows** (blk: str = "$", blk1: str = "$", blk2: str = "$", ldrblk: str = ") → None
Set arrows by block names or AutoCAD standard arrow names, set DIMTSZ to 0 which disables tick.

Parameters
- **blk** – block/arrow name for both arrows, if DIMSAH is 0
- **blk1** – block/arrow name for first arrow, if DIMSAH is 1
- **blk2** – block/arrow name for second arrow, if DIMSAH is 1
- **ldrblk** – block/arrow name for leader

**set_tick** (size: float = 1) → None
Set tick size, which also disables arrows, a tick is just an oblique stroke as marker.

Parameters **size** – arrow size in drawing units

**set_text_align** (halign: str = None, valign: str = None, vshift: float = None) → None
Set measurement text alignment, halign defines the horizontal alignment (requires DXF R2000+), valign defines the vertical alignment, above1 and above2 means above extension line 1 or 2 and aligned with extension line.

Parameters
- **halign** – “left”, “right”, “center”, “above1”, “above2”, requires DXF R2000+
- **valign** – “above”, “center”, “below”
- **vshift** – vertical text shift, if valign is “center”; >0 shift upward, <0 shift downwards

Set dimension text format, like prefix and postfix string, rounding rule and number of decimal places.

Parameters
- **prefix** – Dimension text prefix text as string
- **postfix** – Dimension text postfix text as string
- **rnd** – Rounds all dimensioning distances to the specified value, for instance, if DIMRND is set to 0.25, all distances round to the nearest 0.25 unit. If you set DIMRND to 1.0, all distances round to the nearest integer.
- **dec** – Sets the number of decimal places displayed for the primary units of a dimension, requires DXF R2000+
- **sep** – “.” or “,” as decimal separator, requires DXF R2000+
- **leading_zeros** – Suppress leading zeros for decimal dimensions if False
- **trailing_zeros** – Suppress trailing zeros for decimal dimensions if False

6.8. Reference
**set_dimline_format** *(color: int = None, linetype: str = None, lineweight: int = None, extension: float = None, disable1: bool = None, disable2: bool = None)*

Set dimension line properties

**Parameters**

- **color** – color index
- **linetype** – linetype as string, requires DXF R2007+
- **lineweight** – line weight as int, 13 = 0.13mm, 200 = 2.00mm, requires DXF R2000+
- **extension** – extension length
- **disable1** – True to suppress first part of dimension line, requires DXF R2000+
- **disable2** – True to suppress second part of dimension line, requires DXF R2000+

**set_extline_format** *(color: int = None, lineweight: int = None, extension: float = None, offset: float = None, fixed_length: float = None)*

Set common extension line attributes.

**Parameters**

- **color** – color index
- **lineweight** – line weight as int, 13 = 0.13mm, 200 = 2.00mm
- **extension** – extension length above dimension line
- **offset** – offset from measurement point
- **fixed_length** – set fixed length extension line, length below the dimension line

**set_extline1** *(linetype: str = None, disable=False)*

Set extension line 1 attributes.

**Parameters**

- **linetype** – linetype for extension line 1, requires DXF R2007+
- **disable** – disable extension line 1 if True

**set_extline2** *(linetype: str = None, disable=False)*

Set extension line 2 attributes.

**Parameters**

- **linetype** – linetype for extension line 2, requires DXF R2007+
- **disable** – disable extension line 2 if True

**set_tolerance** *(upper: float, lower: float = None, hfactor: float = 1.0, align: str = None, dec: int = None, leading_zeros: bool = None, trailing_zeros: bool = None)* → None

Set tolerance text format, upper and lower value, text height factor, number of decimal places or leading and trailing zero suppression.

**Parameters**

- **upper** – upper tolerance value
- **lower** – lower tolerance value, if None same as upper
- **hfactor** – tolerance text height factor in relation to the dimension text height
- **align** – tolerance text alignment “TOP”, “MIDDLE”, “BOTTOM”, requires DXF R2000+
- **dec** – Sets the number of decimal places displayed, requires DXF R2000+
• **leading_zeros** – suppress leading zeros for decimal dimensions if `False`, requires DXF R2000+
• **trailing_zeros** – suppress trailing zeros for decimal dimensions if `False`, requires DXF R2000+

**set_limits** *(upper: float, lower: float, hfactor: float = 1.0, dec: int = None, leading_zeros: bool = None, trailing_zeros: bool = None) → None*

Set limits text format, upper and lower limit values, text height factor, number of decimal places or leading and trailing zero suppression.

**Parameters**

- **upper** – upper limit value added to measurement value
- **lower** – lower lower value subtracted from measurement value
- **hfactor** – limit text height factor in relation to the dimension text height
- **dec** – Sets the number of decimal places displayed, requires DXF R2000+
- **leading_zeros** – suppress leading zeros for decimal dimensions if `False`, requires DXF R2000+
- **trailing_zeros** – suppress trailing zeros for decimal dimensions if `False`, requires DXF R2000+

**VPort**

The viewport table ([DXF Reference](https://ezdxf.readthedocs.io/en/latest/dxfref/) stores the modelspace viewport configurations. So this entries just modelspace viewports, not paperspace viewports, for paperspace viewports see the `Viewport` entity.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th><code>ezdxf.entities.DXFEntity</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td><code>'VPORT'</code></td>
</tr>
<tr>
<td>Factory function</td>
<td><code>Drawing.viewports.new()</code></td>
</tr>
</tbody>
</table>

See also:

DXF Internals: `VPORT Configuration Table`

class `ezdxf.entities.VPort`

```python
Subclass of `DXFEntity`

Defines a viewport configurations for the modelspace.

```python
dxf.owner
    Handle to owner(`ViewportTable`).

dxf.name
    Viewport name

dxf.flags
    Standard flag values (bit-coded values):

| 16 | If set, table entry is externally dependent on an xref |
| 32 | If both this bit and bit 16 are set, the externally dependent xref has been successfully resolved |
| 64 | If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is only for the benefit of AutoCAD) |
```
dxlf. lower_left
    Lower-left corner of viewport

dxlf. upper_right
    Upper-right corner of viewport

dxlf. center
    View center point (in DCS)

dxlf. snap_base
    Snap base point (in DCS)

dxlf. snap_spacing
    Snap spacing X and Y

dxlf. grid_spacing
    Grid spacing X and Y

dxlf. direction_point
    View direction from target point (in WCS)

dxlf. target_point
    View target point (in WCS)

dxlf. height
    View height

dxlf. aspect_ratio

dxlf. lens_length
    Lens focal length in mm

dxlf. front_clipping
    Front clipping plane (offset from target point)

dxlf. back_clipping
    Back clipping plane (offset from target point)

dxlf. snap_rotation
    Snap rotation angle in degrees

dxlf. view_twist
    View twist angle in degrees

dxlf. status

dxlf. view_mode

dxlf. circle_zoom

dxlf. fast_zoom

dxlf. ucs_icon

dxlf. snap_on

dxlf. grid_on

dxlf. snap_style

dxlf. snap_isopair
**View**

The View table (DXF Reference) stores named views of the model or paperspace layouts. This stored views makes parts of the drawing or some view points of the model in a CAD applications more accessible. This views have no influence to the drawing content or to the generated output by exporting PDFs or plotting on paper sheets, they are just for the convenience of CAD application users.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFEntity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'VIEW'</td>
</tr>
<tr>
<td>Factory function</td>
<td>Drawing.views.new()</td>
</tr>
</tbody>
</table>

See also:

DXF Internals: *VIEW Table*

```python
class ezdxf.entities.View

    dxf.owner
        Handle to owner (*Table*).
    dxf.name
        Name of view.
    dxf.flags
        Standard flag values (bit-coded values):

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>If set, this is a paper space view</td>
</tr>
<tr>
<td>16</td>
<td>If set, table entry is externally dependent on an xref</td>
</tr>
<tr>
<td>32</td>
<td>If both this bit and bit 16 are set, the externally dependent xref has been successfully resolved</td>
</tr>
<tr>
<td>64</td>
<td>If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is only for the benefit of AutoCAD)</td>
</tr>
</tbody>
</table>

    dxf.height
        View height (in DCS)
    dxf.width
        View width (in DCS)
    dxf.center_point
        View center point (in DCS)
    dxf.direction_point
        View direction from target (in WCS)
    dxf.target_point
        Target point (in WCS)
    dxf.lens_length
        Lens length
    dxf.front_clipping
        Front clipping plane (offset from target point)
    dxf.back_clipping
        Back clipping plane (offset from target point)
    dxf.view_twist
        Twist angle in degrees.```
**dxf.view_mode**

View mode (see VIEWMODE system variable)

**dxf.render_mode**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2D Optimized (classic 2D)</td>
</tr>
<tr>
<td>1</td>
<td>Wireframe</td>
</tr>
<tr>
<td>2</td>
<td>Hidden line</td>
</tr>
<tr>
<td>3</td>
<td>Flat shaded</td>
</tr>
<tr>
<td>4</td>
<td>Gouraud shaded</td>
</tr>
<tr>
<td>5</td>
<td>Flat shaded with wireframe</td>
</tr>
<tr>
<td>6</td>
<td>Gouraud shaded with wireframe</td>
</tr>
</tbody>
</table>

**dxf.ucs**

1 if there is a UCS associated to this view; 0 otherwise

**dxf.ucs_origin**

UCS origin as (x, y, z) tuple (appears only if ucs is set to 1)

**dxf.ucs_xaxis**

UCS x-axis as (x, y, z) tuple (appears only if ucs is set to 1)

**dxf.ucs_yaxis**

UCS y-axis as (x, y, z) tuple (appears only if ucs is set to 1)

**dxf.ucs_ortho_type**

Orthographic type of UCS (appears only if ucs is set to 1)

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UCS is not orthographic</td>
</tr>
<tr>
<td>1</td>
<td>Top</td>
</tr>
<tr>
<td>2</td>
<td>Bottom</td>
</tr>
<tr>
<td>3</td>
<td>Front</td>
</tr>
<tr>
<td>4</td>
<td>Back</td>
</tr>
<tr>
<td>5</td>
<td>Left</td>
</tr>
<tr>
<td>6</td>
<td>Right</td>
</tr>
</tbody>
</table>

**dxf.elevation**

UCS elevation

**dxf.ucs_handle**

Handle of UCSTable if UCS is a named UCS. If not present, then UCS is unnamed (appears only if ucs is set to 1)

**dxf.base_ucs_handle**

Handle of UCSTable of base UCS if UCS is orthographic. If not present and ucs_ortho_type is non-zero, then base UCS is taken to be WORLD (appears only if ucs is set to 1)

**dxf.camera_plottable**

1 if the camera is plottable

**dxf.background_handle**

Handle to background object (optional)

**dxf.live_selection_handle**

Handle to live section object (optional)
dxf.visual_style_handle
Handle to visual style object (optional)

dxf.sun_handle
Sun hard ownership handle.

AppID

Defines an APPID (DXF Reference). These table entries maintain a set of names for all registered applications.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFEntity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'APPID'</td>
</tr>
<tr>
<td>Factory function</td>
<td>Drawing.appids.new()</td>
</tr>
</tbody>
</table>

class ezdxf.entities.AppID

dxf.owner
Handle to owner (Table).

dxf.name
User-supplied (or application-supplied) application name (for extended data).

dxf.flags
Standard flag values (bit-coded values):

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>If set, table entry is externally dependent on an xref</td>
</tr>
<tr>
<td>32</td>
<td>If both this bit and bit 16 are set, externally dependent xref has been successfully resolved</td>
</tr>
<tr>
<td>64</td>
<td>If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is only for the benefit of AutoCAD)</td>
</tr>
</tbody>
</table>

UCS

Defines an named or unnamed user coordinate system (DXF Reference) for usage in CAD applications. This UCS table entry does not interact with ezdxf in any way, to do coordinate transformations by ezdxf use the ezdxf.math.UCS class.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFEntity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'UCS'</td>
</tr>
<tr>
<td>Factory function</td>
<td>Drawing.ucs.new()</td>
</tr>
</tbody>
</table>

See also:

UCS and OCS

class ezdxf.entities.UCSTable

dxf.owner
Handle to owner (Table).

dxf.name
UCS name (str).
**dxflags**

Standard flags (bit-coded values):

<table>
<thead>
<tr>
<th>Bit Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>If set, table entry is externally dependent on an xref</td>
</tr>
<tr>
<td>32</td>
<td>If both this bit and bit 16 are set, the externally dependent xref has been successfully resolved</td>
</tr>
<tr>
<td>64</td>
<td>If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is only for the benefit of AutoCAD)</td>
</tr>
</tbody>
</table>

**dxorigin**

Origin as \((x, y, z)\) tuple

**dxaxis**

- X-axis direction as \((x, y, z)\) tuple
- Y-axis direction as \((x, y, z)\) tuple

**ucs() \(\rightarrow\) UCS**

Returns an `ezdxf.math.UCS` object for this UCS table entry.

**BlockRecord**

`BLOCK_RECORD` (DXF Reference) is the core management structure for `BlockLayout` and `Layout`. This is an internal DXF structure managed by `ezdxf`, package users don’t have to care about it.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th><code>ezdxf.entities.DXFEntity</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'BLOCK_RECORD'</td>
</tr>
<tr>
<td>Factory function</td>
<td><code>Drawing.block_records.new()</code></td>
</tr>
</tbody>
</table>

class `ezdxf.entities.BlockRecord`

- **dxf.owner**
  
  Handle to owner (`Table`).

- **dxf.name**
  
  Name of associated BLOCK.

- **dxf.layout**
  
  Handle to associated `DXFLayout`, if paperspace layout or modelspace else 0

- **dxf.explode**
  
  1 for BLOCK references can be exploded else 0

- **dxf.scale**
  
  1 for BLOCK references can be scaled else 0

- **dxf.units**
  
  BLOCK insert units
<table>
<thead>
<tr>
<th>Unit</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unitless</td>
<td>0</td>
</tr>
<tr>
<td>Inches</td>
<td>1</td>
</tr>
<tr>
<td>Feet</td>
<td>2</td>
</tr>
<tr>
<td>Miles</td>
<td>3</td>
</tr>
<tr>
<td>Millimeters</td>
<td>4</td>
</tr>
<tr>
<td>Centimeters</td>
<td>5</td>
</tr>
<tr>
<td>Meters</td>
<td>6</td>
</tr>
<tr>
<td>Kilometers</td>
<td>7</td>
</tr>
<tr>
<td>Microinches</td>
<td>8</td>
</tr>
<tr>
<td>Mils</td>
<td>9</td>
</tr>
<tr>
<td>Yards</td>
<td>10</td>
</tr>
<tr>
<td>Angstroms</td>
<td>11</td>
</tr>
<tr>
<td>Nanometers</td>
<td>12</td>
</tr>
<tr>
<td>Microns</td>
<td>13</td>
</tr>
<tr>
<td>Decimeters</td>
<td>14</td>
</tr>
<tr>
<td>Decameters</td>
<td>15</td>
</tr>
<tr>
<td>Hectometers</td>
<td>16</td>
</tr>
<tr>
<td>Gigameters</td>
<td>17</td>
</tr>
<tr>
<td>Astronomical units</td>
<td>18</td>
</tr>
<tr>
<td>Light years</td>
<td>19</td>
</tr>
<tr>
<td>Parsecs</td>
<td>20</td>
</tr>
<tr>
<td>US Survey Feet</td>
<td>21</td>
</tr>
<tr>
<td>US Survey Inch</td>
<td>22</td>
</tr>
<tr>
<td>US Survey Yard</td>
<td>23</td>
</tr>
<tr>
<td>US Survey Mile</td>
<td>24</td>
</tr>
</tbody>
</table>

**is_active_paperspace**
True if is “active” paperspace layout.

**is_any_paperspace**
True if is any kind of paperspace layout.

**is_any_layout**
True if is any kind of modelspace or paperspace layout.

**is_block_layout**
True if not any kind of modelspace or paperspace layout, just a regular block definition.

**is_modelspace**
True if is the modelspace layout.

**Internal Structure**

Do not change this structures, this is just an information for experienced developers!

The BLOCK_RECORD is the owner of all the entities in a layout and stores them in an *EntitySpace* object (BlockRecord.entity_space). For each layout exist a BLOCK definition in the BLOCKS section, a reference to the *Block* entity is stored in BlockRecord.block.

*Modelspace* and *Paperspace* layouts require an additional *DXFLayout* object in the OBJECTS section.

See also:

More information about *Block Management Structures* and *Layout Management Structures*.

6.8. Reference
Blocks

A block definition (BlockLayout) is a collection of DXF entities, which can be placed multiply times at different layouts or other blocks as references to the block definition.

See also:
Tutorial for Blocks and DXF Internals: Block Management Structures

Block

BLOCK (DXF Reference) entity is embedded into the BlockLayout object. The BLOCK entity is accessible by the BlockLayout.block attribute.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFEntity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'BLOCK'</td>
</tr>
<tr>
<td>Factory function</td>
<td>Drawing.blocks.new() (returns a BlockLayout)</td>
</tr>
</tbody>
</table>

See also:
Tutorial for Blocks and DXF Internals: Block Management Structures

class ezdxf.entities.Block

dxf.handle
BLOCK handle as plain hex string. (feature for experts)

dxf.owner
Handle to owner as plain hex string. (feature for experts)

dxf.layer
Layer name as string; default value is '0'

dxf.name
BLOCK name as string. (case insensitive)

dxf.base_point
BLOCK base point as (x, y, z) tuple, default value is (0, 0, 0)

Insertion location referenced by the Insert entity to place the block reference and also the center of rotation and scaling.

dxf.flags
BLOCK flags (bit-coded)

<table>
<thead>
<tr>
<th>1</th>
<th>Anonymous block generated by hatching, associative dimensioning, other internal operations, or an application</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Block has non-constant attribute definitions (this bit is not set if the block has any attribute definitions that are constant, or has no attribute definitions at all)</td>
</tr>
<tr>
<td>4</td>
<td>Block is an external reference (xref)</td>
</tr>
<tr>
<td>8</td>
<td>Block is an xref overlay</td>
</tr>
<tr>
<td>16</td>
<td>Block is externally dependent</td>
</tr>
<tr>
<td>32</td>
<td>This is a resolved external reference, or dependent of an external reference (ignored on input)</td>
</tr>
<tr>
<td>64</td>
<td>This definition is a referenced external reference (ignored on input)</td>
</tr>
</tbody>
</table>
\texttt{dxf.xref\_path}

File system path as string, if this block defines an external reference (XREF).

\textbf{is\_layout\_block}

Returns True if this is a Modelspace or Paperspace block definition.

\textbf{is\_anonymous}

Returns True if this is an anonymous block generated by hatching, associative dimensioning, other internal operations, or an application.

\textbf{is\_xref}

Returns True if block is an external referenced file.

\textbf{is\_xref\_overlay}

Returns True if block is an external referenced overlay file.

\textbf{EndBlk}

ENDBLK entity is embedded into the \texttt{BlockLayout} object. The ENDBLK entity is accessible by the \texttt{BlockLayout.endblk} attribute.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFEntity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'ENDBLK'</td>
</tr>
</tbody>
</table>

\textbf{class} ezdxf.entities.EndBlk

\textbf{dxf.handle}

BLOCK handle as plain hex string. (feature for experts)

\textbf{dxf.owner}

Handle to owner as plain hex string. (feature for experts)

\textbf{dxf.layer}

Layer name as string; should always be the same as \texttt{Block.dxf.layer}

\textbf{Insert}

Block reference (DXF Reference) with maybe attached attributes (Attrib).

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'INSERT'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_blockref()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
</tbody>
</table>

See also:

\textit{Tutorial for Blocks}

\textbf{Warning:} Do not instantiate entity classes by yourself - always use the provided factory functions!

TODO: influence of layer, linetype, color DXF attributes to block entities

6.8. Reference
class ezdxf.entities.Insert

dxf.name
    BLOCK name (str)

dxf.insert
    Insertion location of the BLOCK base point as (2D/3D Point in OCS)

dxf.xscale
    Scale factor for x direction (float)

dxf.yscale
    Scale factor for y direction (float)
    Not all CAD applications support non-uniform scaling (e.g. LibreCAD).

dxf.zscale
    Scale factor for z direction (float)
    Not all CAD applications support non-uniform scaling (e.g. LibreCAD).

dxf.rotation
    Rotation angle in degrees (float)

dxf.row_count
    Count of repeated insertions in row direction, MINSERT entity if > 1 (int)

dxf.row_spacing
    Distance between two insert points (MINSERT) in row direction (float)

dxf.column_count
    Count of repeated insertions in column direction, MINSERT entity if > 1 (int)

dxf.column_spacing
    Distance between two insert points (MINSERT) in column direction (float)

attribs
    A list of all attached Attrib entities.

has_scaling
    Returns True if any axis scaling is applied.

has_uniform_scaling
    Returns True if scaling is uniform in x-, y- and z-axis ignoring reflections e.g. (1, 1, -1) is uniform scaling.

mcount
    Returns the multi-insert count, MINSERT (multi-insert) processing is required if mcount > 1.
    New in version 0.14.

set_scale(factor: float)
    Set uniform scaling.

block() → Optional[BlockLayout]
    Returns associated BlockLayout.

place(insert: Vertex = None, scale: Tuple[float, float, float] = None, rotation: float = None) → Insert
    Set block reference placing location insert, scaling and rotation attributes. Parameters which are None will not be altered.

    Parameters
        • insert – insert location as (x, y [,z]) tuple
• **scale** – (x-scale, y-scale, z-scale) tuple
• **rotation** – rotation angle in degrees

**grid** *(size: Tuple[int, int] = (1, 1), spacing: Tuple[float, float] = (1, 1)) → Insert*

Place block reference in a grid layout, grid **size** defines the row- and column count, spacing defines the distance between two block references.

**Parameters**
• **size** – grid size as (row_count, column_count) tuple
• **spacing** – distance between placing as (row_spacing, column_spacing) tuple

**has_attrib** *(tag: str, search_const: bool = False) → bool*

Returns True if ATTRIB tag exist, for search_const doc see `get_attrib()`.

**Parameters**
• **tag** – tag name as string
• **search_const** – search also const ATTDEF entities

**get_attrib** *(tag: str, search_const: bool = False) → Union[Attrib, AttDef, None]*

Get attached **Attrib** entity with dxf.tag == tag, returns None if not found. Some applications may not attach constant ATTRIB entities, set search_const to True, to get at least the associated AttDef entity.

**Parameters**
• **tag** – tag name
• **search_const** – search also const ATTDEF entities

**get_attrib_text** *(tag: str, default: str = None, search_const: bool = False) → str*

Get content text of attached **Attrib** entity with dxf.tag == tag, returns default if not found. Some applications may not attach constant ATTRIB entities, set search_const to True, to get content text of the associated AttDef entity.

**Parameters**
• **tag** – tag name
• **default** – default value if ATTRIB tag is absent
• **search_const** – search also const ATTDEF entities

**add_attrib** *(tag: str, text: str, insert: Vertex = (0, 0), dxfattribs: dict = None) → Attrib*

Attach an **Attrib** entity to the block reference.

Example for appending an attribute to an INSERT entity with none standard alignment:

```python
e.add_attrib('EXAMPLETAG', 'example text').set_pos((3, 7), align='MIDDLE_CENTER')
```

**Parameters**
• **tag** – tag name as string
• **text** – content text as string
• **insert** – insert location as tuple (x, y[, z]) in WCS
• **dxfattribs** – additional DXF attributes for the ATTRIB entity
add_auto_attribs (values: Dict[str, str]) → ezdxf.entities.insert.Insert

Attach for each Attdef entity, defined in the block definition, automatically an Attrib entity to the block reference and set tag/value DXF attributes of the ATTRIB entities by the key/value pairs (both as strings) of the values dict. The ATTRIB entities are placed relative to the insert location of the block reference, which is identical to the block base point.

This method avoids the wrapper block of the add_auto_blockref() method, but the visual results may not match the results of CAD applications, especially for non uniform scaling. If the visual result is very important to you, use the add_auto_blockref() method.

Parameters values – Attrib tag values as tag/value pairs

delete_attrib (tag: str, ignore=False) → None

Delete an attached Attrib entity from INSERT. If ignore is False, an DXFKeyError exception is raised, if ATTRIB tag does not exist.

Parameters

• tag – ATTRIB name
• ignore – False for raising DXFKeyError if ATTRIB tag does not exist.

Raises DXFKeyError – if ATTRIB tag does not exist.

delete_all_attribs () → None

Delete all Attrib entities attached to the INSERT entity.

reset_transformation () → None

Reset block reference parameters location, rotation and extrusion vector.

transform (m: Matrix44) → Insert

Transform INSERT entity by transformation matrix m inplace.

Unlike the transformation matrix m, the INSERT entity can not represent a non orthogonal target coordinate system, for this case an InsertTransformationError will be raised.

translate (dx: float, dy: float, dz: float) → Insert

Optimized INSERT translation about dx in x-axis, dy in y-axis and dz in z-axis.

virtual_entities (skipped_entity_callback: Callable[[DXFGraphic, str], None] = None) → Iterable[DXFGraphic]

Yields “virtual” entities of a block reference. This method is meant to examine the block reference entities at the “exploded” location without really “exploding” the block reference. The ‘skipped_entity_callback()’ will be called for all entities which are not processed, signature: skipped_entity_callback(entity: DXFEntity, reason: str), entity is the original (untransformed) DXF entity of the block definition, the reason string is an explanation why the entity was skipped.

This entities are not stored in the entity database, have no handle and are not assigned to any layout. It is possible to convert this entities into regular drawing entities by adding the entities to the entities database and a layout of the same DXF document as the block reference:

doc.entitydb.add(entity)
msp = doc.modelspace()
msp.add_entity(entity)

This method does not resolve the MININSERT attributes, only the sub-entities of the base INSERT will be returned. To resolve MININSERT entities check if multi insert processing is required, that’s the case if property Insert.mcount > 1, use the Insert.multi_insert () method to resolve the MININSERT entity into single INSERT entities.
Parameters **skipped_entity_callback** – called whenever the transformation of an entity is not supported and so was skipped

**multi_insert** () → Iterable[Insert]
Yields a virtual INSERT entity for each grid element of a MINSERT entity (multi-insert).
New in version 0.14.

**explode** (target_layout: BaseLayout = None) → EntityQuery
Explode block reference entities into target layout, if target layout is None, the target layout is the layout of the block reference. This method destroys the source block reference entity.

Transforms the block entities into the required WCS location by applying the block reference attributes `insert`, `extrusion`, `rotation` and the scaling values `xscale`, `yscale` and `zscale`.

Attached ATTRIB entities are converted to TEXT entities, this is the behavior of the BURST command of the AutoCAD Express Tools.

Returns an `EntityQuery` container with all “exploded” DXF entities.

**Warning:** Non uniform scaling may lead to incorrect results for text entities (TEXT, MTEXT, ATTRIB) and maybe some other entities.

Parameters **target_layout** – target layout for exploded entities, `None` for same layout as source entity.

**ucs** () → UCS
Returns the block reference coordinate system as `ezdxf.math.UCS` object.

### Attrib

The ATTRIB (DXF Reference) entity represents a text value associated with a tag. In most cases an ATTRIB is appended to an `Insert` entity, but it can also appear as standalone entity.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th><code>ezdxf.entities.Text</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'ATTRIB'</td>
</tr>
<tr>
<td>Factory function</td>
<td><code>ezdxf.layouts.BaseLayout.add_attrib()</code> (stand alone entity)</td>
</tr>
<tr>
<td>Factory function</td>
<td><code>Insert.add_attrib()</code> (attached to Insert)</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td><code>Common graphical DXF attributes</code></td>
</tr>
</tbody>
</table>

See also:

*Tutorial for Blocks*

**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!

class `ezdxf.entities.Attrib`
ATTrib supports all DXF attributes and methods of parent class `Text`. 

6.8. Reference
dxf.tag
Tag to identify the attribute (str)

dxf.text
Attribute content as text (str)

is_invisible
Attribute is invisible (does not appear).

is_const
This is a constant attribute.

is_verify
Verification is required on input of this attribute. (CAD application feature)

is_preset
No prompt during insertion. (CAD application feature)

AttDef

The ATTDEF (DXF Reference) entity is a template in a BlockLayout, which will be used to create an attached Attrib entity for an Insert entity.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'ATTDEF'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_attdef()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
</tbody>
</table>

See also:

Tutorial for Blocks

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.Attdef
ATTDEF supports all DXF attributes and methods of parent class Text.

dxf.tag
Tag to identify the attribute (str)

dxf.text
Attribute content as text (str)

dxf.prompt
Attribute prompt string. (CAD application feature)

dxf.field_length
Just relevant to CAD programs for validating user input

is_invisible
Attribute is invisible (does not appear).

is_const
This is a constant attribute.

is_verify
Verification is required on input of this attribute. (CAD application feature)
**is_preset**
No prompt during insertion. (CAD application feature)

---

**Layouts**

**Layout Manager**

The layout manager is unique to each DXF drawing, access the layout manager as `layouts` attribute of the `Drawing` object (e.g. `doc.layouts.rename("Layout1", "PlanView")`).

```python
class ezdxf.layouts.Layouts
    The Layouts class manages Paperspace layouts and the Modelspace.
    __len__() -> int
        Returns count of existing layouts, including the modelspace layout.
    __contains__(name: str) -> bool
        Returns True if layout name exist.
    __iter__() -> Iterable[Layout]
        Returns iterable of all layouts as Layout objects, including the modelspace layout.
    names() -> List[str]
        Returns a list of all layout names, all names in original case sensitive form.
    names_in_taborder() -> List[str]
        Returns all layout names in tab order as shown in CAD applications.
    modelspace() -> Modelspace
        Returns the Modelspace layout.
    get(name: str) -> Layout
        Returns Layout by name, case insensitive “Model” == “MODEL”.
        Parameters name -- layout name as shown in tab, e.g. 'Model' for modelspace
    new(name: str, dxfattribs: dict = None) -> Paperspace
        Returns a new Paperspace layout.
        Parameters
            • name -- layout name as shown in tabs in CAD applications
            • dxfattribs -- additional DXF attributes for the DXFLayout entity
        Raises
            • DXFValueError -- Invalid characters in layout name.
            • DXFValueError -- Layout name already exist.
    rename(old_name: str, new_name: str) -> None
        Rename a layout from old_name to new_name. Can not rename layout 'Model' and the new name of a layout must not exist.
        Parameters
            • old_name -- actual layout name, case insensitive
            • new_name -- new layout name, case insensitive
        Raises
            • DXFValueError -- try to rename 'Model'
```

---

6.8. Reference
• DXFValueError – Layout new_name already exist.

**delete** *(name: str) → None*
Delete layout name and destroy all entities in that layout.

**Parameters**
- **name** *(str)* – layout name as shown in tabs

**Raises**
- DXFKeyError – if layout name do not exists
- DXFValueError – deleting modelspace layout is not possible
- DXFValueError – deleting last paperspace layout is not possible

**active_layout** *(()) → Paperspace*
Returns the active paperspace layout.

**set_active_layout** *(name: str) → None*
Set layout name as active paperspace layout.

**get_layout_for_entity** *(entity: DXFEntity) → Layout*
Returns the owner layout for a DXF entity.

**Layout Types**

A Layout represents and manages DXF entities, there are three different layout objects:

- **Modelspace** is the common working space, containing basic drawing entities.
- **Paperspace** is arrangement of objects for printing and plotting, this layout contains basic drawing entities and viewports to the Modelspace.
- **BlockLayout** works on an associated Block, Blocks are collections of drawing entities for reusing by block references.

**Warning:** Do not instantiate layout classes by yourself - always use the provided factory functions!

**Entity Ownership**

A layout owns all entities residing in their entity space, this means the dxf.owner attribute of any DXFGraphic in this layout is the dxf.handle of the layout, and deleting an entity from a layout is the end of life of this entity, because it is also deleted from the EntityDB. But it is possible to just unlink an entity from a layout, so it can be assigned to another layout, use the move_to_layout() method to move entities between layouts.

**BaseLayout**

class ezdxf.layouts.BaseLayout
BaseLayout is the common base class for Layout and BlockLayout.

- **is_alive**
  - False if layout is deleted.

- **is_active_paperspace**
  - True if is active layout.
is_any_paperspace
True if is any kind of paperspace layout.

is_modelspace
True if is modelspace layout.

is_any_layout
True if is any kind of modelspace or paperspace layout.

is_block_layout
True if not any kind of modelspace or paperspace layout, just a regular block definition.

units
set drawing units.

Type Get/Set layout/block drawing units as enum, see also

Type ref

__len__() → int
Returns count of entities owned by the layout.

__iter__() → Iterable[DXFGraphic]
Returns iterable of all drawing entities in this layout.

__getitem__(index)
Get entity at index.

The underlying data structure for storing entities is organized like a standard Python list, therefore index can be any valid list indexing or slicing term, like a single index layout[-1] to get the last entity, or an index slice layout[:10] to get the first 10 or less entities as List[DXFGraphic].

get_extension_dict() → ExtensionDict
Returns the associated extension dictionary, creates a new one if necessary.

delete_entity(entity: DXFGraphic) → None
Delete entity from layout entity space and the entity database, this destroys the entity.

delete_all_entities() → None
Delete all entities from this layout and from entity database, this destroys all entities in this layout.

unlink_entity(entity: DXFGraphic) → None
Unlink entity from layout but does not delete entity from the entity database, this removes entity just from the layout entity space.

query(query: str = '*') → EntityQuery
Get all DXF entities matching the Entity Query String.

groupby(dxfattrib: str = '', key: KeyFunc = None) → dict
Returns a dict of entity lists, where entities are grouped by a dxfattrib or a key function.

Parameters
- dxfattrib – grouping by DXF attribute like 'layer'
- key – key function, which accepts a DXFGraphic entity as argument and returns the grouping key of an entity or None to ignore the entity. Reason for ignoring: a queried DXF attribute is not supported by entity.

move_to_layout(entity: DXFGraphic, layout: BaseLayout) → None
Move entity to another layout.

Parameters
- entity – DXF entity to move
• **layout** – any layout (modelspace, paperspace, block) from same drawing

**add_entity** *(entity: DXFGraphic) → None*

Add an existing DXFGraphic entity to a layout, but be sure to unlink *(unlink_entity())* entity from the previous owner layout. Adding entities from a different DXF drawing is not supported.

**add_foreign_entity** *(entity: DXFGraphic, copy=True) → None*

Add a foreign DXF entity to a layout, this foreign entity could be from another DXF document or an entity without an assigned DXF document. The intention of this method is to add simple entities from another DXF document or from a DXF iterator, for more complex operations use the importer add-on. Especially objects with BLOCK section (INSERT, DIMENSION, MLEADER) or OBJECTS section dependencies (IMAGE, UNDERLAY) can not be supported by this simple method.

Not all DXF types are supported and every dependency or resource reference from another DXF document will be removed except attribute layer will be preserved but only with default attributes like color 7 and linetype CONTINUOUS because the layer attribute doesn’t need a layer table entry.

If the entity is part of another DXF document, it will be unlinked from this document and its entity database if argument *copy* is False, else the entity will be copied. Unassigned entities like from DXF iterators will just be added.

Supported DXF types:

- POINT
- LINE
- CIRCLE
- ARC
- ELLIPSE
- LWPOLYLINE
- SPLINE
- POLYLINE
- 3DFACE
- SOLID
- TRACE
- SHAPE
- MESH
- ATTRIB
- ATTDEF
- TEXT
- MTEXT
- HATCH

**Parameters**

- **entity** – DXF entity to copy or move
- **copy** – if True copy entity from other document else unlink from other document
add_point (location: Vertex, dxfattribs: Dict[KT, VT] = None) → Point
Add a Point entity at location.

Parameters

• location – 2D/3D point in WCS
• dxfattribs – additional DXF attributes

add_line (start: Vertex, end: Vertex, dxfattribs: Dict[KT, VT] = None) → Line
Add a Line entity from start to end.

Parameters

• start – 2D/3D point in WCS
• end – 2D/3D point in WCS
• dxfattribs – additional DXF attributes

add_circle (center: Vertex, radius: float, dxfattribs: Dict[KT, VT] = None) → Circle
Add a Circle entity. This is an 2D element, which can be placed in space by using OCS.

Parameters

• center – 2D/3D point in WCS
• radius – circle radius
• dxfattribs – additional DXF attributes

add_ellipse (center: Vertex, major_axis: Vertex = (1, 0, 0), ratio: float = 1, start_param: float = 0, end_param: float = 6.283185307179586, dxfattribs: Dict[KT, VT] = None) → Ellipse
Add an Ellipse entity, ratio is the ratio of minor axis to major axis, start_param and end_param defines start and end point of the ellipse, a full ellipse goes from 0 to 2*pi. The ellipse goes from start to end param in counter clockwise direction.

Parameters

• center – center of ellipse as 2D/3D point in WCS
• major_axis – major axis as vector (x, y, z)
• ratio – ratio of minor axis to major axis in range +/-[1e-6, 1.0]
• start_param – start of ellipse curve
• end_param – end param of ellipse curve
• dxfattribs – additional DXF attributes

add_arc (center: Vertex, radius: float, start_angle: float, end_angle: float, is_counter_clockwise: bool = True, dxfattribs: Dict[KT, VT] = None) → Arc
Add an Arc entity. The arc goes from start_angle to end_angle in counter clockwise direction by default, set parameter is_counter_clockwise to False for clockwise orientation.

Parameters

• center – center of arc as 2D/3D point in WCS
• radius – arc radius
• start_angle – start angle in degrees
• end_angle – end angle in degrees
• is_counter_clockwise – False for clockwise orientation
• dxfattribs – additional DXF attributes
add_solid(points: Iterable[Vertex], dxfattribs: Dict[KT, VT] = None) → Solid
Add a Solid entity, points is an iterable of 3 or 4 points.

Parameters

- **points** – iterable of 3 or 4 2D/3D points in WCS
- **dxfattribs** – additional DXF attributes for Solid entity

add_trace(points: Iterable[Vertex], dxfattribs: Dict[KT, VT] = None) → Trace
Add a Trace entity, points is an iterable of 3 or 4 points.

Parameters

- **points** – iterable of 3 or 4 2D/3D points in WCS
- **dxfattribs** – additional DXF attributes for Trace entity

add_3dface(points: Iterable[Vertex], dxfattribs: Dict[KT, VT] = None) → Face3d
Add a 3DFace entity, points is an iterable 3 or 4 2D/3D points.

Parameters

- **points** – iterable of 3 or 4 2D/3D points in WCS
- **dxfattribs** – additional DXF attributes for 3DFace entity

add_text(text: str, dxfattribs: Dict[KT, VT] = None) → Text
Add a Text entity, see also Style.

Parameters

- **text** – content string
- **dxfattribs** – additional DXF attributes for Text entity

add_blockref(name: str, insert: Vertex, dxfattribs: Dict[KT, VT] = None) → Insert
Add an Insert entity.

When inserting a block reference into the modelspace or another block layout with different units, the scaling factor between these units should be applied as scaling attributes (xscale, ...) e.g. modelspace in meters and block in centimeters, xscale has to be 0.01.

Parameters

- **name** – block name as str
- **insert** – insert location as 2D/3D point in WCS
- **dxfattribs** – additional DXF attributes for Insert entity

add_auto_blockref(name: str, insert: Vertex, values: Dict[str, str], dxfattribs: Dict[KT, VT] = None) → Insert
Add an Insert entity. This method adds for each Attdef entity, defined in the block definition, automatically an Attrib entity to the block reference and set tag/value DXF attributes of the ATTRIB entities by the key/value pairs (both as strings) of the values dict.

The Attrib entities are placed relative to the insert point, which is equal to the block base point.

This method wraps the INSERT and all the ATTRIB entities into an anonymous block, which produces the best visual results, especially for non uniform scaled block references, because the transformation and scaling is done by the CAD application. But this makes evaluation of block references with attributes more complicated, if you prefer INSERT and ATTRIB entities without a wrapper block use the add_blockref_with_attribs() method.

Parameters
• \textbf{name} – block name
• \textbf{insert} – insert location as 2D/3D point in \textit{WCS}
• \textbf{values} – \texttt{Attrib} tag values as tag/value pairs
• \textbf{dxfattribs} – additional DXF attributes for \texttt{Insert} entity

\begin{verbatim}
add_attrib(tag: str, text: str, insert: Vertex = (0, 0), dxfattribs: Dict[KT, VT] = None) → Attrib

Add an \texttt{Attrib} as stand alone DXF entity.
Parameters
• \textbf{tag} – tag name as string
• \textbf{text} – tag value as string
• \textbf{insert} – insert location as 2D/3D point in \textit{WCS}
• \textbf{dxfattribs} – additional DXF attributes for \texttt{Attrib} entity
\end{verbatim}

\begin{verbatim}
add_attdef(tag: str, insert: Vertex = (0, 0), text: str = '', dxfattribs: Dict[KT, VT] = None) → AttDef

Add an \texttt{AttDef} as stand alone DXF entity.
Set position and alignment by the idiom:

\begin{verbatim}
layout.add_attdef('NAME').set_pos((2, 3), align='MIDDLE_CENTER')
\end{verbatim}

Parameters
• \textbf{tag} – tag name as string
• \textbf{insert} – insert location as 2D/3D point in \textit{WCS}
• \textbf{text} – tag value as string
• \textbf{dxfattribs} – additional DXF attributes
\end{verbatim}

\begin{verbatim}
add_polyline2d(points: Iterable[Vertex], format: str = None, *, close: bool = False, dxfattribs: Dict[KT, VT] = None) → Polyline

Add a 2D \texttt{Polyline} entity.
Parameters
• \textbf{points} – iterable of 2D points in \textit{WCS}
• \textbf{close} – True for a closed polyline
• \textbf{format} – user defined point format like \texttt{add_lwpolyline()}, default is None
• \textbf{dxfattribs} – additional DXF attributes
\end{verbatim}

\begin{verbatim}
add_polyline3d(points:Iterable[Vertex], *, close: bool = False, dxfattribs: Dict[KT, VT] = None) → Polyline

Add a 3D \texttt{Polyline} entity.
Parameters
• \textbf{points} – iterable of 3D points in \textit{WCS}
• \textbf{close} – True for a closed polyline
• \textbf{dxfattribs} – additional DXF attributes
\end{verbatim}

\begin{verbatim}
add_polymesh(size: Tuple[int, int] = (3, 3), dxfattribs: Dict[KT, VT] = None) → Polymesh

Add a \texttt{Polymesh} entity, which is a wrapper class for the \texttt{POLYLINE} entity. A polymesh is a grid of \texttt{mcount x ncount} vertices and every vertex has its own (x, y, z)-coordinates.
\end{verbatim}
Parameters

• **size** – 2-tuple \((mcount, ncount)\)

• **dxfattribs** – additional DXF attributes for **Polyline** entity

**add_polyface** \((dxfattribs: Dict[KT, VT] = None) \rightarrow \text{Polyface}\)

Add a **Polyface** entity, which is a wrapper class for the POLYLINE entity.

Parameters **dxfattribs** – additional DXF attributes for **Polyline** entity

**add_shape** \((name: str, insert: Vertex = (0, 0), size: float = 1.0, dxfattribs: Dict[KT, VT] = None) \rightarrow \text{Shape}\)

Add a **Shape** reference to a external stored shape.

Parameters

• **name** – shape name as string

• **insert** – insert location as 2D/3D point in **WCS**

• **size** – size factor

• **dxfattribs** – additional DXF attributes

**add_lwpolyline** \((points: Iterable[Vertex], format: str = \text{'xyseb'}, *, close: bool = \text{False}, dxfattribs: Dict[KT, VT] = None) \rightarrow \text{LWPolyline}\)

Add a 2D polyline as **LWPolyline** entity. A points are defined as \((x, y, [\text{start}_width, [\text{end}_width, [\text{bulge}]]])\) tuples, but order can be redefined by the **format** argument. Set \text{start}_width, \text{end}_width to 0 to be ignored like \((x, y, 0, 0, \text{bulge})\).

The **LWPolyline** is defined as a single DXF entity and needs less disk space than a **Polyline** entity. (requires DXF R2000)

Format codes:

• **x** = x-coordinate

• **y** = y-coordinate

• **s** = start width

• **e** = end width

• **b** = bulge value

• **v** = \((x, y, [z])\) tuple \((z\text{-axis is ignored})\)

Parameters

• **points** – iterable of \((x, y, [\text{start}_width, [\text{end}_width, [\text{bulge}]]])\) tuples

• **format** – user defined point format, default is "xyseb"

• **close** – True for a closed polyline

• **dxfattribs** – additional DXF attributes

**add_mtext** \((text: str, dxfattribs: Dict[KT, VT] = None) \rightarrow \text{MText}\)

Add a multiline text entity with automatic text wrapping at boundaries as **MText** entity. (requires DXF R2000)

Parameters

• **text** – content string

• **dxfattribs** – additional DXF attributes
add_ray(start: Vertex, unit_vector: Vertex, dxfattribs: Dict[KT, VT] = None) → Ray
Add a Ray that begins at start point and continues to infinity (construction line). (requires DXF R2000)

Parameters
- **start** – location 3D point in WCS
- **unit_vector** – 3D vector (x, y, z)
- **dxfattribs** – additional DXF attributes

add_xline(start: Vertex, unit_vector: Vertex, dxfattribs: Dict[KT, VT] = None) → XLine
Add an infinity XLine (construction line). (requires DXF R2000)

Parameters
- **start** – location 3D point in WCS
- **unit_vector** – 3D vector (x, y, z)
- **dxfattribs** – additional DXF attributes

Add a MLine entity

Parameters
- **vertices** – MLINE vertices (in WCS)
- **close** – True to add a closed MLINE
- **dxfattribs** – additional DXF attributes for MLine entity

add_spline(fit_points: Iterable[Vertex] = None, degree: int = 3, dxfattribs: Dict[KT, VT] = None) → Spline
Add a B-spline (Spline entity) defined by the given fit_points - the control points and knot values are created by the CAD application, therefore it is not predictable how the rendered spline will look like, because for every set of fit points exists an infinite set of B-splines.

If fit_points is None, an “empty” spline will be created, all data has to be set by the user.

The SPLINE entity requires DXF R2000.

AutoCAD creates a spline through fit points by a global curve interpolation and an unknown method to estimate the direction of the start- and end tangent.

See also:
- Tutorial for Spline
- ezdxf.math.fit_points_to_cad_cv()

Parameters
- **fit_points** – iterable of fit points as (x, y[, z]) in WCS, creates an empty Spline if None
- **degree** – degree of B-spline, max. degree supported by AutoCAD is 11
- **dxfattribs** – additional DXF attributes

add_cad_spline_control_frame(fit_points: Iterable[Vertex], tangents: Iterable[Vertex] = None, estimate: str = '5-p', dxfattribs: Dict[KT, VT] = None) → Spline
Add a Spline entity passing through the given fit points. This method tries to create the same curve
as CAD applications do. To understand the limitations and for more information see function `ezdxf.math.fit_points_to_cad_cv()`.

Parameters

- **fit_points** – iterable of fit points as (x, y[, z]) in WCS
- **tangents** – start- and end tangent, default is autodetect
- **estimate** – tangent direction estimation method
- **dxfattribs** – additional DXF attributes

```python
def add_spline_control_frame(fit_points: Iterable[Vertex], degree: int = 3, method: str = 'chord', dxfattribs: Dict[KT, VT] = None) -> Spline
```

Add a Spline entity passing through the given fit_points, the control points are calculated by a global curve interpolation without start- and end tangent constrains. The new Spline entity is defined by control points and not by the fit points, therefore the Spline looks always the same, no matter which CAD application renders the Spline.

- “uniform”: creates a uniform t vector, from 0 to 1 evenly spaced, see uniform method
- “distance”, “chord”: creates a t vector with values proportional to the fit point distances, see chord length method
- “centripetal”, “sqrt_chord”: creates a t vector with values proportional to the fit point sqrt(distances), see centripetal method
- “arc”: creates a t vector with values proportional to the arc length between fit points.

Use function `add_cad_spline_control_frame()` to create Spline entities from fit points similar to CAD application including start- and end tangent constraints.

Parameters

- **fit_points** – iterable of fit points as (x, y[, z]) in WCS
- **degree** – degree of B-spline, max. degree supported by AutoCAD is 11
- **method** – calculation method for parameter vector t
- **dxfattribs** – additional DXF attributes

```python
def add_open_spline(control_points: Iterable[Vertex], degree: int = 3, knots: Iterable[float] = None, dxfattribs: Dict[KT, VT] = None) -> Spline
```

Add an open uniform Spline defined by control_points. (requires DXF R2000)

Open uniform B-splines start and end at your first and last control point.

Parameters

- **control_points** – iterable of 3D points in WCS
- **degree** – degree of B-spline, max. degree supported by AutoCAD is 11
- **knots** – knot values as iterable of floats
- **dxfattribs** – additional DXF attributes

```python
def add_rational_spline(control_points: Iterable[Vertex], weights: Sequence[float], degree: int = 3, knots: Iterable[float] = None, dxfattribs: Dict[KT, VT] = None) -> Spline
```

Add an open rational uniform Spline defined by control_points. (requires DXF R2000)

weights has to be an iterable of floats, which defines the influence of the associated control point to the shape of the B-spline, therefore for each control point is one weight value required.

Open rational uniform B-splines start and end at the first and last control point.
Parameters

- **control_points** – iterable of 3D points in *WCS*
- **weights** – weight values as iterable of floats
- **degree** – degree of B-spline, max. degree supported by AutoCAD is 11
- **knots** – knot values as iterable of floats
- **dxfattribs** – additional DXF attributes

```python
add_hatch(color: int = 7, dxfattribs: Dict[KT, VT] = None) \rightarrow Hatch
```

Add a *Hatch* entity. (requires DXF R2007)

Parameters

- **color** – ACI (AutoCAD Color Index), default is 7 (black/white).
- **dxfattribs** – additional DXF attributes

```python
add_mesh(dxfattribs: Dict[KT, VT] = None) \rightarrow Mesh
```

Add a *Mesh* entity. (requires DXF R2007)

Parameters **dxfattribs** – additional DXF attributes

```python
add_image(image_def: ImageDef, insert: Vertex, size_in_units: Tuple[float, float], rotation: float = 0.0, dxfattribs: Dict[KT, VT] = None) \rightarrow Image
```

Add an *Image* entity, requires a *ImageDef* entity, see Tutorial for Image and ImageDef. (requires DXF R2000)

Parameters

- **image_def** – required image definition as *ImageDef*
- **insert** – insertion point as 3D point in *WCS*
- **size_in_units** – size as *(x, y)* tuple in drawing units
- **rotation** – rotation angle around the extrusion axis, default is the z-axis, in degrees
- **dxfattribs** – additional DXF attributes

```python
add_wipeout(vertices: Iterable[Vertex], dxfattribs: Dict[KT, VT] = None) \rightarrow Wipeout
```

Add a *ezdxf.entities.Wipeout* entity, the masking area is defined by WCS *vertices*.

This method creates only a 2D entity in the xy-plane of the layout, the z-axis of the input vertices are ignored.

```python
add_underlay(underlay_def: UnderlayDefinition, insert: Vertex = (0, 0, 0), scale=(1, 1, 1), rotation: float = 0.0, dxfattribs: Dict[KT, VT] = None) \rightarrow Underlay
```

Add an *Underlay* entity, requires a *UnderlayDefinition* entity, see Tutorial for Underlay and UnderlayDefinition. (requires DXF R2000)

Parameters

- **underlay_def** – required underlay definition as *UnderlayDefinition*
- **insert** – insertion point as 3D point in *WCS*
- **scale** – underlay scaling factor as *(x, y, z)* tuple or as single value for uniform scaling for x, y and z
- **rotation** – rotation angle around the extrusion axis, default is the z-axis, in degrees
- **dxfattribs** – additional DXF attributes

Add horizontal, vertical and rotated Dimension line. If an UCS is used for dimension line rendering, all point definitions in UCS coordinates, translation into WCS and OCS is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default. See also: Tutorial for Linear Dimensions

This method returns a DimStyleOverride object - to create the necessary dimension geometry, you have to call render() manually, this two step process allows additional processing steps on the Dimension entity between creation and rendering.

Note: ezdxf ignores some DIMSTYLE variables, so render results may differ from BricsCAD or AutoCAD.

Parameters

- **base** – location of dimension line, any point on the dimension line or its extension will do (in UCS)
- **p1** – measurement point 1 and start point of extension line 1 (in UCS)
- **p2** – measurement point 2 and start point of extension line 2 (in UCS)
- **location** – user defined location for text mid point (in UCS)
- **text** – None or "<>" the measurement is drawn as text, " " (one space) suppresses the dimension text, everything else text is drawn as dimension text
- **dimstyle** – dimension style name (DimStyle table entry), default is 'EZDXF'
- **angle** – angle from ucs/wcs x-axis to dimension line in degrees
- **text_rotation** – rotation angle of the dimension text as absolute angle (x-axis=0, y-axis=90) in degrees
- **override** – DimStyleOverride attributes
- **dxfattribs** – additional DXF attributes for Dimension entity

Returns: DimStyleOverride


Add multiple linear dimensions for iterable points. If an UCS is used for dimension line rendering, all point definitions in UCS coordinates, translation into WCS and OCS is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default. See also: Tutorial for Linear Dimensions

This method sets many design decisions by itself, the necessary geometry will be generated automatically, no required nor possible render() call. This method is easy to use but you get what you get.

Note: ezdxf ignores some DIMSTYLE variables, so render results may differ from BricsCAD or AutoCAD.

Parameters
• **base** – location of dimension line, any point on the dimension line or its extension will do (in UCS)

• **points** – iterable of measurement points (in UCS)

• **angle** – angle from ucs/wcs x-axis to dimension line in degrees (0 = horizontal, 90 = vertical)

• **ucs** – user defined coordinate system

• **avoid_double_rendering** – suppresses the first extension line and the first arrow if possible for continued dimension entities

• **dimstyle** – dimension style name (DimStyle table entry), default is 'EZDXF'

• **override** – DimStyleOverride attributes

• **dxfattribs** – additional DXF attributes for Dimension entity

• **discard** – discard rendering result for friendly CAD applications like BricsCAD to get a native and likely better rendering result. (does not work with AutoCAD)

```python
add_aligned_dim(p1: Vertex, p2: Vertex, distance: float, text: str = '<>', dimstyle: str = 'EZDXF',
override: Dict[KT, VT] = None, dxfattribs: Dict[KT, VT] = None) → DimStyleOverride
```

Add linear dimension aligned with measurement points p1 and p2. If an **UCS** is used for dimension line rendering, all point definitions in UCS coordinates, translation into **WCS** and **OCS** is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default. See also: **Tutorial for Linear Dimensions**

This method returns a DimStyleOverride object, to create the necessary dimension geometry, you have to call DimStyleOverride.render() manually, this two step process allows additional processing steps on the Dimension entity between creation and rendering.

**Note:** ezdxf ignores some DIMSTYLE variables, so render results may differ from BricsCAD or AutoCAD.

### Parameters

• **p1** – measurement point 1 and start point of extension line 1 (in UCS)

• **p2** – measurement point 2 and start point of extension line 2 (in UCS)

• **distance** – distance of dimension line from measurement points

• **text** – None or “<>” the measurement is drawn as text, “ ” (one space) suppresses the dimension text, everything else **text** is drawn as dimension text

• **dimstyle** – dimension style name (DimStyle table entry), default is 'EZDXF'

• **override** – DimStyleOverride attributes

• **dxfattribs** – DXF attributes for Dimension entity

**Returns:** DimStyleOverride

```python
add_radius_dim(center: Vertex, mpoint: Vertex = None, radius: float = None, angle: float = None,
location: Vertex = None, text: str = '<>', dimstyle: str = 'EZ_RADIUS', override:
Dict[KT, VT] = None, dxfattribs: Dict[KT, VT] = None) → DimStyleOverride
```

Add a radius Dimension line. The radius dimension line requires a center point and a point **mpoint** on
the circle or as an alternative a \textit{radius} and a dimension line \textit{angle} in degrees. See also: \textit{Tutorial for Radius Dimensions}

If an \textit{UCS} is used for dimension line rendering, all point definitions in UCS coordinates, translation into \textit{WCS} and \textit{OCS} is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default.

This method returns a \texttt{DimStyleOverride} object - to create the necessary dimension geometry, you have to call \texttt{render()} manually, this two step process allows additional processing steps on the Dimension entity between creation and rendering.

Following render types are supported:

- Default text location outside: text aligned with dimension line; dimension style: '\texttt{EZ_RADIUS}'
- Default text location outside horizontal: '\texttt{EZ_RADIUS}' + dimtoh=1
- Default text location inside: text aligned with dimension line; dimension style: '\texttt{EZ_RADIUS_INSIDE}'
- Default text location inside horizontal: '\texttt{EZ_RADIUS_INSIDE}' + dimtih=1
- User defined text location: argument \textit{location} \texttt{!= None}, text aligned with dimension line; dimension style: '\texttt{EZ_RADIUS}'
- User defined text location horizontal: argument \textit{location} \texttt{!= None}, '\texttt{EZ_RADIUS}' + dimtoh=1 for text outside horizontal, '\texttt{EZ_RADIUS}' + dimtih=1 for text inside horizontal

Placing the dimension text at a user defined \textit{location}, overrides the \textit{mpoint} and the \textit{angle} argument, but requires a given \textit{radius} argument. The \textit{location} argument does not define the exact text location, instead it defines the dimension line starting at \textit{center} and the measurement text midpoint projected on this dimension line going through \textit{location}, if text is aligned to the dimension line. If text is horizontal, \textit{location} is the kink point of the dimension line from radial to horizontal direction.

\textbf{Note:} \textit{ezdxf} ignores some DIMSTYLE variables, so render results may differ from BricsCAD or AutoCAD.

\textbf{Parameters}

- \texttt{center} – center point of the circle (in UCS)
- \texttt{mpoint} – measurement point on the circle, overrides \textit{angle} and \textit{radius} (in UCS)
- \texttt{radius} – radius in drawing units, requires argument \textit{angle}
- \texttt{angle} – specify angle of dimension line in degrees, requires argument \textit{radius}
- \texttt{location} – user defined dimension text location, overrides \textit{mpoint} and \textit{angle}, but requires \textit{radius} (in UCS)
- \texttt{text} – \texttt{None} or "<>" the measurement is drawn as text, " " (one space) suppresses the dimension text, everything else \texttt{text} is drawn as dimension text
- \texttt{dimstyle} – dimension style name (\texttt{DimStyle} table entry), default is '\texttt{EZ_RADIUS}'
- \texttt{override} – \texttt{DimStyleOverride} attributes
- \texttt{dxfattribs} – additional DXF attributes for Dimension entity

\textbf{Returns:} \texttt{DimStyleOverride}
add_radius_dim_2p(center: Vertex, mpoint: Vertex, text: str = '<>', dimstyle: str = 'EZ_RADIUS',
override: Dict[KT, VT] = None, dxfattribs: Dict[KT, VT] = None) → DimStyleOverride
Shortcut method to create a radius dimension by center point, measurement point on the circle and the measurement text at the default location defined by the associated dimstyle, for further information see general method add_radius_dim().

- dimstyle 'EZ_RADIUS': places the dimension text outside
- dimstyle 'EZ_RADIUS_INSIDE': places the dimension text inside

Parameters

- **center** – center point of the circle (in UCS)
- **mpoint** – measurement point on the circle (in UCS)
- **text** – None or "<>" the measurement is drawn as text, " " (one space) suppresses the dimension text, everything else text is drawn as dimension text
- **dimstyle** – dimension style name (DimStyle table entry), default is 'EZ_RADIUS'
- **override** – DimStyleOverride attributes
- **dxfattribs** – additional DXF attributes for Dimension entity

Returns: DimStyleOverride

add_radius_dim_cra(center: Vertex, radius: float, angle: float, text: str = '<>', dimstyle: str = 'EZ_RADIUS',
override: Dict[KT, VT] = None, dxfattribs: Dict[KT, VT] = None) → DimStyleOverride
Shortcut method to create a radius dimension by (c)enter point, (r)adius and (a)ngle, the measurement text is placed at the default location defined by the associated dimstyle, for further information see general method add_radius_dim().

- dimstyle 'EZ_RADIUS': places the dimension text outside
- dimstyle 'EZ_RADIUS_INSIDE': places the dimension text inside

Parameters

- **center** – center point of the circle (in UCS)
- **radius** – radius in drawing units
- **angle** – angle of dimension line in degrees
- **text** – None or "<>" the measurement is drawn as text, " " (one space) suppresses the dimension text, everything else text is drawn as dimension text
- **dimstyle** – dimension style name (DimStyle table entry), default is 'EZ_RADIUS'
- **override** – DimStyleOverride attributes
- **dxfattribs** – additional DXF attributes for Dimension entity

Returns: DimStyleOverride

add_diameter_dim(center: Vertex, mpoint: Vertex = None, radius: float = None, angle: float = None,
location: Vertex = None, text: str = '<>', dimstyle: str = 'EZ_RADIUS',
override: Dict[KT, VT] = None, dxfattribs: Dict[KT, VT] = None) → DimStyleOverride
Add a diameter Dimension line. The diameter dimension line requires a center point and a point mpoint on the circle or as an alternative a radius and a dimension line angle in degrees.
If an UCS is used for dimension line rendering, all point definitions in UCS coordinates, translation into WCS and OCS is done by the rendering function. Extrusion vector is defined by UCS or \((0, 0, 1)\) by default.

This method returns a DimStyleOverride object - to create the necessary dimension geometry, you have to call render() manually, this two step process allows additional processing steps on the Dimension entity between creation and rendering.

**Note:** ezdxf ignores some DIMSTYLE variables, so render results may differ from BricsCAD or AutoCAD.

### Parameters

- **center** – specifies the center of the circle (in UCS)
- **mpoint** – specifies the measurement point on the circle (in UCS)
- **radius** – specify radius, requires argument angle, overrides p1 argument
- **angle** – specify angle of dimension line in degrees, requires argument radius, overrides p1 argument
- **location** – user defined location for text mid point (in UCS)
- **text** – None or "<>" the measurement is drawn as text, " " (one space) suppresses the dimension text, everything else text is drawn as dimension text
- **dimstyle** – dimension style name (DimStyle table entry), default is 'EZ_RADIUS'
- **override** – DimStyleOverride attributes
- **dxfattribs** – additional DXF attributes for Dimension entity

**Returns:** DimStyleOverride

```python
```

Shortcut method to create a diameter dimension by two points on the circle and the measurement text at the default location defined by the associated dimstyle, for further information see general method add_diameter_dim(). Center point of the virtual circle is the mid point between p1 and p2.

- dimstyle 'EZ_RADIUS': places the dimension text outside
- dimstyle 'EZ_RADIUS_INSIDE': places the dimension text inside

### Parameters

- **p1** – first point of the circle (in UCS)
- **p2** – second point on the opposite side of the center point of the circle (in UCS)
- **text** – None or "<>" the measurement is drawn as text, " " (one space) suppresses the dimension text, everything else text is drawn as dimension text
- **dimstyle** – dimension style name (DimStyle table entry), default is 'EZ_RADIUS'
- **override** – DimStyleOverride attributes
- **dxfattribs** – additional DXF attributes for Dimension entity

**Returns:** DimStyleOverride
add_leader(\texttt{vertices: Iterable[Vertex], dimstyle: str = 'EZDXF', override: Dict[KT, VT] = None, dxfattribs: Dict[KT, VT] = None} \rightarrow \texttt{Leader})

The Leader entity represents an arrow, made up of one or more vertices (or spline fit points) and an arrowhead. The label or other content to which the Leader is attached is stored as a separate entity, and is not part of the Leader itself. (requires DXF R2000)

Leader shares its styling infrastructure with Dimension.

By default a Leader without any annotation is created. For creating more fancy leaders and annotations see documentation provided by Autodesk or Demystifying DXF: LEADER and MULTILEADER implementation notes.

Parameters

- \texttt{vertices} – leader vertices (in WCS)
- \texttt{dimstyle} – dimension style name (DimStyle table entry), default is 'EZDXF'
- \texttt{override} – override DimStyleOverride attributes
- \texttt{dxfattribs} – additional DXF attributes for Leader entity

add_body(\texttt{acis_data: str = None, dxfattribs: Dict[KT, VT] = None} \rightarrow \texttt{Body})

Add a Body entity. (requires DXF R2000)

Parameters

- \texttt{acis_data} – ACIS data as iterable of text lines as strings, no interpretation by ezdxf possible
- \texttt{dxfattribs} – additional DXF attributes

add_region(\texttt{acis_data: str = None, dxfattribs: Dict[KT, VT] = None} \rightarrow \texttt{Region})

Add a Region entity. (requires DXF R2000)

Parameters

- \texttt{acis_data} – ACIS data as iterable of text lines as strings, no interpretation by ezdxf possible
- \texttt{dxfattribs} – additional DXF attributes

add_3dsolid(\texttt{acis_data: str = None, dxfattribs: Dict[KT, VT] = None} \rightarrow \texttt{Solid3d})

Add a 3DSOLID entity (Solid3d). (requires DXF R2000)

Parameters

- \texttt{acis_data} – ACIS data as iterable of text lines as strings, no interpretation by ezdxf possible
- \texttt{dxfattribs} – additional DXF attributes

add_surface(\texttt{acis_data: str = None, dxfattribs: Dict[KT, VT] = None} \rightarrow \texttt{Surface})

Add a Surface entity. (requires DXF R2007)

Parameters

- \texttt{acis_data} – ACIS data as iterable of text lines as strings, no interpretation by ezdxf possible
- \texttt{dxfattribs} – additional DXF attributes

add_extruded_surface(\texttt{acis_data: str = None, dxfattribs: Dict[KT, VT] = None} \rightarrow \texttt{ExtrudedSurface})

Add a ExtrudedSurface entity. (requires DXF R2007)

Parameters
- **acis_data** – ACIS data as iterable of text lines as strings, no interpretation by ezdxf possible

- **dxfattribs** – additional DXF attributes

**add_lofted_surface** (acis_data: str = None, dxfattribs: Dict[KT, VT] = None) → LoftedSurface
Add a LoftedSurface entity. (requires DXF R2007)

**Parameters**
- **acis_data** – ACIS data as iterable of text lines as strings, no interpretation by ezdxf possible
- **dxfattribs** – additional DXF attributes

**add_revolved_surface** (acis_data: str = None, dxfattribs: Dict[KT, VT] = None) → RevolvedSurface
Add a RevolvedSurface entity. (requires DXF R2007)

**Parameters**
- **acis_data** – ACIS data as iterable of text lines as strings, no interpretation by ezdxf possible
- **dxfattribs** – additional DXF attributes

**add_swept_surface** (acis_data: str = None, dxfattribs: Dict[KT, VT] = None) → SweptSurface
Add a SweptSurface entity. (requires DXF R2007)

**Parameters**
- **acis_data** – ACIS data as iterable of text lines as strings, no interpretation by ezdxf possible
- **dxfattribs** – additional DXF attributes

### Layout

class ezdxf.layouts.Layout

*Layout* is a subclass of *BaseLayout* and common base class of *Modelspace* and *Paperspace*.

- **name**
  Layout name as shown in tabs of *CAD* applications.

- **dxf**
  Returns the DXF name space attribute of the associated *DXFLayout* object.
  This enables direct access to the underlying LAYOUT entity, e.g. `Layout.dxf.layout_flags`

- **contains**
  Returns True if *entity* is stored in this layout.

- **reset_extents**
  Reset extents to given values or the AutoCAD default values.
  “Drawing extents are the bounds of the area occupied by objects.” (Quote Autodesk Knowledge Network)

**Parameters**
- **extmin** – minimum extents or (+1e20, +1e20, +1e20) as default value
- **extmax** – maximum extents or (-1e20, -1e20, -1e20) as default value
reset_limits \((\text{limmin} = \text{None}, \text{limmax} = \text{None}) \rightarrow \text{None}\)

Reset limits to given values or the AutoCAD default values.

“Sets an invisible rectangular boundary in the drawing area that can limit the grid display and limit clicking or entering point locations.” (Quote Autodesk Knowledge Network)

The Paperspace class has an additional method \(\text{reset_paper_limits()}\) to deduce the default limits from the paper size settings.

**Parameters**

- **\text{extmin}** – minimum extents or \((0, 0)\) as default
- **\text{extmax}** – maximum extents or \((\text{paper width}, \text{paper height})\) as default value

\(\text{set_plot_type}(\text{value}: \text{int} = 5) \rightarrow \text{None}\)

<table>
<thead>
<tr>
<th>0</th>
<th>last screen display</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>drawing extents</td>
</tr>
<tr>
<td>2</td>
<td>drawing limits</td>
</tr>
<tr>
<td>3</td>
<td>view specific (defined by Layout.dxf.plot_view_name)</td>
</tr>
<tr>
<td>4</td>
<td>window specific (defined by Layout.set_plot_window_limits())</td>
</tr>
<tr>
<td>5</td>
<td>layout information (default)</td>
</tr>
</tbody>
</table>

**Parameters** \text{value} – plot type

**Raises** DXFValueError – for \text{value} out of range

\(\text{set_plot_style}(\text{name}: \text{str} = \text{'ezdxf.ctb'}, \text{show}: \text{bool} = \text{False}) \rightarrow \text{None}\)

Set plot style file of type \text{.ctb}.

**Parameters**

- **\text{name}** – plot style filename
- **\text{show}** – show plot style effect in preview? (AutoCAD specific attribute)

\(\text{set_plot_window}(\text{lower_left}: \text{Tuple}[\text{float}, \text{float}] = (0, 0), \text{upper_right}: \text{Tuple}[\text{float}, \text{float}] = (0, 0)) \rightarrow \text{None}\)

Set plot window size in (scaled) paper space units.

**Parameters**

- **\text{lower_left}** – lower left corner as 2D point
- **\text{upper_right}** – upper right corner as 2D point

\(\text{set_redraw_order}(\text{handles}: \text{Union}[\text{Dict}[\text{KT}, \text{VT}], \text{Iterable}[\text{Tuple}[\text{str}, \text{str}]]]) \rightarrow \text{None}\)

If the header variable $\text{SORTENTS Regen}$ flag (bit-code value 16) is set, AutoCAD regenerates entities in ascending handles order.

To change redraw order associate a different sort handle to entities, this redefines the order in which the entities are regenerated. \text{handles} can be a dict of entity_handle and sort_handle as \((k, v)\) pairs, or an iterable of (entity_handle, sort_handle) tuples.

The sort_handle doesn’t have to be unique, some or all entities can share the same sort handle and a sort handle can be an existing handle.

The “0” handle can be used, but this sort_handle will be drawn as latest (on top of all other entities) and not as first as expected.
Parameters **handles** – iterable or dict of handle associations; an iterable of 2-tuples (entity_handle, sort_handle) or a dict (k, v) association as (entity_handle, sort_handle)

**get_redraw_order()** → Iterable[Tuple[str, str]]
Returns iterable for all existing table entries as (entity_handle, sort_handle) pairs, see also **set_redraw_order()**.

**plot_viewport_borders**(state: bool = True) → None

**show_plot_styles**(state: bool = True) → None

**plot_centered**(state: bool = True) → None

**plot_hidden**(state: bool = True) → None

**use_standard_scale**(state: bool = True) → None

**use_plot_styles**(state: bool: bool = True) → None

**scale_lineweights**(state: bool = True) → None

**print_lineweights**(state: bool = True) → None

**draw_viewports_first**(state: bool = True) → None

**model_type**(state: bool = True) → None

**update_paper**(state: bool = True) → None

**zoom_to_paper_on_update**(state: bool = True) → None

**plot_flags_initializing**(state: bool = True) → None

**prev_plot_init**(state: bool = True) → None

**set_plot_flags**(flag, state: bool = True) → None

---

**ModeSpace**

class ezdxf.layouts.Modelspace

**Modelspace** is a subclass of **Layout**.

The modelspace contains the “real” world representation of the drawing subjects in real world units.

**name**
Name of modelspace is fixed as “Model”.

**new_geodata**(dxfattribs: dict = None) → GeoData
Creates a new GeoData entity and replaces existing ones. The GEODATA entity resides in the OBJECTS section and not in the modelspace, it is linked to the modelspace by an ExtensionDict located in BLOCK_RECORD of the modelspace.

The GEODATA entity requires DXF R2010. The DXF reference does not document if other layouts than the modelspace supports geo referencing, so I assume getting/setting geo data may only make sense for the modelspace.

**Parameters** dxfattribs – DXF attributes for GeoData entity

**get_geodata**( ) → Optional[GeoData]
Returns the GeoData entity associated to the modelspace or None.
Paperspace

class ezdxf.layouts.Paperspace

Paperspace is a subclass of Layout.

Paperspace layouts are used to create different drawing sheets of the modelspace subjects for printing or PDF export.

name

Layout name as shown in tabs of CAD applications.

page_setup (size=(297, 210), margins=(10, 15, 10, 15), units='mm', offset=(0, 0), rotation=0, scale=16, name='ezdxf', device='DWG to PDF .pc3')

Setup plot settings and paper size and reset viewports. All parameters in given units (mm or inch).

Reset paper limits, extents and viewports.

Parameters

• size – paper size as (width, height) tuple
• margins – (top, right, bottom, left) hint: clockwise
• units – “mm” or “inch”
• offset – plot origin offset is 2D point
• rotation – see table Rotation
• scale – integer in range [0, 32] defines a standard scale type or as tuple(numerator, denominator) e.g. (1, 50) for scale 1:50
• name – paper name prefix “([name]_[width]_[height]_[unit])”
• device – device .pc3 configuration file or system printer name

<table>
<thead>
<tr>
<th>int</th>
<th>Rotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no rotation</td>
</tr>
<tr>
<td>1</td>
<td>90 degrees counter-clockwise</td>
</tr>
<tr>
<td>2</td>
<td>upside-down</td>
</tr>
<tr>
<td>3</td>
<td>90 degrees clockwise</td>
</tr>
</tbody>
</table>

viewports () → List[Viewport]

Get all VIEWPORT entities defined in this paperspace layout. Returns a list of Viewport objects, sorted by id, the first entity is always the main viewport with an id of 1.

main_viewport () → Optional[Viewport]

Returns the main viewport of this paper space layout, or None if no main viewport exist.

add_viewport (center: Vertex, size: Tuple[float, float], view_center_point: Vertex, view_height: float, dxfattribs: dict = None) → Viewport

Add a new Viewport entity.

reset_viewports () → None

Delete all existing viewports, and create a new main viewport.

reset_main_viewport (center: Vertex = None, size: Vertex = None) → Viewport

Reset the main viewport of this paper space layout to the given values, or reset them to the default values, deduced from the paper settings. Creates a new main viewport if none exist.

Ezdxf does not create a main viewport by default, because CAD applications don’t require one.

Parameters
- **center** – center of the viewport in paper space units
- **size** – viewport size as (width, height) tuple in paper space units

```python
reset_paper_limits() \rightarrow None
```
Set paper limits to default values, all values in paperspace units but without plot scale (?)

```python
get_paper_limits() \rightarrow Tuple[Vec2, Vec2]
```
Returns paper limits in plot paper units, relative to the plot origin.

plot origin = lower left corner of printable area + plot origin offset

**Returns** tuple \((x_1, y_1), (x_2, y_2)\), lower left corner is \((x_1, y_1)\), upper right corner is \((x_2, y_2)\).

### BlockLayout

```python
class ezdxf.layouts.BlockLayout
```

**BlockLayout** is a subclass of **BaseLayout**.
Block layouts are reusable sets of graphical entities, which can be referenced by multiple **Insert** entities. Each reference can be placed, scaled and rotated individually and can have it’s own set of DXF **Attrib** entities attached.

- **name**
  - name of the associated BLOCK and BLOCK_RECORD entities.

- **block**
  - the associated **Block** entity.

- **endblk**
  - the associated **EndBlk** entity.

- **dxf**
  - DXF name space of associated **BlockRecord** table entry.

- **can_explode**
  - Set property to **True** to allow exploding block references of this block.

- **scale_uniformly**
  - Set property to **True** to allow block references of this block only scale uniformly.

- **__contains__**(entity: Union[DXFGraphic, str]) \rightarrow bool
  - Returns **True** if block contains **entity**.

  **Parameters**
  - **entity** – DXFGraphic object or handle as hex string

```python
attdefs() \rightarrow Iterable[AttDef]
```
Returns iterable of all **Attdef** entities.

```python
has_attdef(tag: str) \rightarrow bool
```
Returns **True** if an **Attdef** for **tag** exist.

```python
get_attdef(tag: str) \rightarrow Optional[DXFGraphic]
```
Returns attached **Attdef** entity by **tag** name.

```python
get_attdef_text(tag: str, default: str = None) \rightarrow str
```
Returns text content for **Attdef** **tag** as string or returns **default** if no **Attdef** for **tag** exist.

  **Parameters**
  - **tag** – name of **tag**
Groups

A group is just a bunch of DXF entities tied together. All entities of a group has to be on the same layout (modelspace or any paper layout but not block). Groups can be named or unnamed, but in reality an unnamed groups has just a special name like "*Annnn". The name of a group has to be unique in the drawing. Groups are organized in the main group table, which is stored as attribute groups in the Drawing object.

Group entities have to be in modelspace or any paperspace layout but not in a block definition!

DXFGroup

class ezdxf.entities.dxfgroups.DXFGroup

The group name is not stored in the GROUP entity, it is stored in the GroupCollection object.

dxf.description
    group description (string)

dxf.unnamed
    1 for unnamed, 0 for named group (int)

dxf.selectable
    1 for selectable, 0 for not selectable group (int)

__iter__() → Iterable[ezdxf.entities.dxfentity.DXFEntity]
    Iterate over all DXF entities in DXFGroup as instances of DXFGraphic or inherited (LINE, CIRCLE, ...).

__len__() → int
    Returns the count of DXF entities in DXFGroup.

__getitem__(item)
    Returns entities by standard Python indexing and slicing.

__contains__(item: Union[str, ezdxf.entities.dxfentity.DXFEntity]) → bool
    Returns True if item is in DXFGroup. item has to be a handle string or an object of type DXFEntity or inherited.

handles() → Iterable[str]
    Iterable of handles of all DXF entities in DXFGroup.

edit_data() → List[ezdxf.entities.dxfentity.DXFEntity]
    Context manager which yields all the group entities as standard Python list:

    ```python
    with group.edit_data() as data:
        # add new entities to a group
        data.append(modelspace.add_line((0, 0), (3, 0)))
        # remove last entity from a group
        data.pop()
    ```

    set_data(entities: Iterable[DXFEntity]) → None
    Set entities as new group content, entities should be an iterable DXFGraphic or inherited (LINE, CIRCLE, ...). Raises DXFValueError if not all entities be on the same layout (modelspace or any paperspace layout but not block)

    extend(entities: Iterable[DXFEntity]) → None
    Add entities to DXFGroup.
clear() → None
Remove all entities from DXFGroup, does not delete any drawing entities referenced by this group.

audit (auditor: Auditor) → None
Remove invalid handles from DXFGroup.

Invalid handles are: deleted entities, not all entities in the same layout or entities in a block layout.

GroupCollection

Each Drawing has one group table, which is accessible by the attribute groups.

class ezdxf.entities.dxfgroups.GroupCollection
Manages all DXFGroup objects of a Drawing.

__len__ () → int
Returns the count of DXF groups.

__iter__ ()
Iterate over all existing groups as (name, group) tuples. name is the name of the group as string and group is an DXFGroup object.

__contains__ (name: str) → bool
Returns True if a group name exist.

get (name: str) → DXFGroup
Returns the group name. Raises DXFKeyError if group name does not exist.

groups () → DXFGroup
Iterable of all existing groups.

new (name: str=None, description: str=", selectable: bool=True) → DXFGroup
Creates a new group. If name is None an unnamed group is created, which has an automatically generated name like “*Annnn”.

Parameters

• name – group name as string
• description – group description as string
• selectable – group is selectable if True

delete (group: Union[DXFGroup, str]) → None
Delete group, group can be an object of type DXFGroup or a group name as string.

clear ()
Delete all groups.

audit (auditor: Auditor) → None
Removes empty groups and invalid handles from all groups.

DXF Entities

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!
DXF Entity Base Class

Common base class for all DXF entities and objects.

**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.DXFEntity

dxf

The DXF attributes namespace:

```python
# set attribute value
def layer = 'MyLayer'

# get attribute value
def linetype = entity.dxf.linetype

# delete attribute
def del entity.dxf.linetype

dxf.handle

DXF handle is a unique identifier as plain hex string like F000. (feature for experts)

dxf.owner

Handle to owner as plain hex string like F000. (feature for experts)

doc

Get the associated Drawing instance.

dxftype() -> str

Get DXF type as string, like LINE for the line entity.

__str__() -> str

Returns a simple string representation.

__repr__() -> str

Returns a simple string representation including the class.

has_dxf_attrib(key: str) -> bool

Returns True if DXF attribute key really exist. Raises DXFAttributeError if key is not an supported DXF attribute.

is_supported_dxf_attrib(key: str) -> bool

Returns True if DXF attrb key is supported by this entity. Does not grant that attribute key really exist.

get_dxf_attrib(key: str, default: Any = None) -> Any

Get DXF attribute key, returns default if key doesn’t exist, or raise DXFValueError if default is DXFValueError and no DXF default value is defined:

```python
layer = entity.get_dxf_attrib("layer")
# same as
layer = entity.dxf.layer
```

Raises DXFAttributeError if key is not an supported DXF attribute.

set_dxf_attrib(key: str, value: Any) -> None

Set new value for DXF attribute key:
entity.set_dxf_attrib("layer", "MyLayer")
# same as
entity.dxf.layer = "MyLayer"

Raises DXFAttributeError if key is not an supported DXF attribute.

def dxf_attribs(key: str = None) → Dict[KT, VT]
    Returns a dict with all existing DXF attributes and their values and exclude all DXF attributes listed in set drop.

def update_dxf_attribs(dxfattribs: Dict[KT, VT]) → None
    Set DXF attributes by a dict like {'layer': 'test', 'color': 4}.

def set_flag_state(flag: int, state: bool = True, name: str = 'flags') → None
    Set binary coded flag of DXF attribute name to 1 (on) if state is True, set flag to 0 (off) if state is False.

def get_flag_state(flag: int, name: str = 'flags') → bool
    Returns True if any flag of DXF attribute is 1 (on), else False. Always check only one flag state at the time.

has_extension_dict
    Returns True if entity has an attached ExtensionDict.

def get_extension_dict() → ExtensionDict
    Returns the existing ExtensionDict.

    Raises AttributeError – extension dict does not exist

new_extension_dict() → ExtensionDict

has_app_data(appid: str) → bool
    Returns True if application defined data for appid exist.

def get_app_data(appid: str) → Tags
    Returns application defined data for appid as iterable of tags.

Parameters
    • appid – application name as defined in the APPID table.

    Raises DXFValueError – no data for appid found

set_app_data(appid: str, tags: Iterable)
    Set application defined data for appid as iterable of tags.

Parameters
    • appid – application name as defined in the APPID table.
    • tags – iterable of (code, value) tuples or DXFTag

def discard_app_data(appid: str)
    Discard application defined data for appid. Does not raise an exception if no data for appid exist.

has_xdata(appid: str) → bool
    Returns True if extended data for appid exist.

def get_xdata(appid: str) → Tags
    Returns extended data for appid.

Parameters
    • appid – application name as defined in the APPID table.
Raises DXFValueError – no extended data for appid found

set_xdata(appid: str, tags: Iterable)
Set extended data for appid as iterable of tags.

Parameters

• appid – application name as defined in the APPID table.
• tags – iterable of (code, value) tuples or DXFTag

discard_xdata(appid: str) → None
Discard extended data for appid. Does not raise an exception if no extended data for appid exist.

has_xdata_list(appid: str, name: str) → bool
Returns True if a tag list name for extended data appid exist.

get_xdata_list(appid: str, name: str) → Tags
Returns tag list name for extended data appid.

Parameters

• appid – application name as defined in the APPID table.
• name – extended data list name

Raises DXFValueError – no extended data for appid found or no data list name not found

set_xdata_list(appid: str, name: str, tags: Iterable)
Set tag list name for extended data appid as iterable of tags.

Parameters

• appid – application name as defined in the APPID table.
• name – extended data list name
• tags – iterable of (code, value) tuples or DXFTag

discard_xdata_list(appid: str, name: str) → None
Discard tag list name for extended data appid. Does not raise an exception if no extended data for appid or no tag list name exist.

replace_xdata_list(appid: str, name: str, tags: Iterable)
Replaces tag list name for existing extended data appid by tags. Appends new list if tag list name do not exist, but raises DXFValueError if extended data appid do not exist.

Parameters

• appid – application name as defined in the APPID table.
• name – extended data list name
• tags – iterable of (code, value) tuples or DXFTag

Raises DXFValueError – no extended data for appid found

has_reactors() → bool
Returns True if entity has reactors.

get_reactors() → List[str]
Returns associated reactors as list of handles.

set_reactors(handles: Iterable[str]) → None
Set reactors as list of handles.
**append_reactor_handle** (*handle: str*) → None
Append *handle* to reactors.

**discard_reactor_handle** (*handle: str*) → None
Discard *handle* from reactors. Does not raise an exception if *handle* does not exist.

## DXF Graphic Entity Base Class

Common base class for all graphical DXF entities.

This entities resides in entity spaces like *Modelspace*, any *Paperspace* or *BlockLayout*.

| Subclass of | ezdxf.entities.DXFEntity |

**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!

### class ezdxf.entities.DXFGraphic

**rgb**
Get/set DXF attribute *dxf.true_color* as *(r, g, b)* tuple, returns *None* if attribute *dxf.true_color* is not set.

```python
def rgb(self):  
    if self.dxf.true_color is not None:  
        return (self.dxf.true_color[0], self.dxf.true_color[1], self.dxf.true_color[2])  
    else:  
        return (None, None, None)
```

```python
def _rgb(self, r, g, b):  
    self.dxf.true_color = (r, g, b)
```

This is the recommend method to get/set RGB values, when ever possible do not use the DXF low level attribute *dxf.true_color*.

**transparency**
Get/set transparency value as float. Value range 0 to 1, where 0 means entity is opaque and 1 means entity is 100% transparent (invisible). This is the recommend method to get/set transparency values, when ever possible do not use the DXF low level attribute *DXFGraphic.dxf.transparency*

This attribute requires DXF R2004 or later, returns 0 for prior DXF versions and raises *DXFAttributeError* for setting *transparency* in older DXF versions.

**ocs** () → OCS
Returns object coordinate system (*OCS*) for 2D entities like *Text* or *Circle*, returns *None* for entities without OCS support.

**get_layout** () → BaseLayout
Returns the owner layout or returns *None* if entity is not assigned to any layout.

**unlink_from_layout** () → None
Unlink entity from associated layout. Does nothing if entity is already unlinked.

It is more efficient to call the *unlink_entity()* method of the associated layout, especially if you have to unlink more than one entity.

**copy_to_layout** (*layout: BaseLayout*) → DXFEntity
Copy entity to another *layout*, returns new created entity as *DXFEntity* object. Copying between different DXF drawings is not supported.

**Parameters**
- *layout* – any layout (model space, paper space, block)

**Raises** *DXFStructureError* – for copying between different DXF drawings
move_to_layout *(layout: BaseLayout, source: BaseLayout=None)*
Move entity from model space or a paper space layout to another layout. For block layout as source, the block layout has to be specified. Moving between different DXF drawings is not supported.

**Parameters**

- **layout** – any layout (model space, paper space, block)
- **source** – provide source layout, faster for DXF R12, if entity is in a block layout

**Raises** DXFStructureError – for moving between different DXF drawings

graphic_properties() → Dict[KT, VT]
Returns the important common properties layer, color, linetype, lineweight, ltscale, true_color and color_name as dxattribs dict.

has_hyperlink() → bool
Returns True if entity has an attached hyperlink.

get_hyperlink() → Tuple[str, str, str]
Returns hyperlink, description and location.

set_hyperlink(link: str, description: str = None, location: str = None)
Set hyperlink of an entity.

transform *(t: Matrix44)* → DXFGraphic
Inplace transformation interface, returns self (floating interface).

**Parameters** m – 4x4 transformation matrix (ezdxf.math.Matrix44)

translate *(dx: float, dy: float, dz: float)* → DXFGraphic
Translate entity inplace about dx in x-axis, dy in y-axis and dz in z-axis, returns self (floating interface).

Basic implementation uses the transform() interface, subclasses may have faster implementations.

scale *(sx: float, sy: float, sz: float)* → DXFGraphic
Scale entity inplace about dx in x-axis, dy in y-axis and dz in z-axis, returns self (floating interface).

scale_uniform *(s: float)* → DXFGraphic
Scale entity inplace uniform about s in x-axis, y-axis and z-axis, returns self (floating interface).

rotate_x *(angle: float)* → DXFGraphic
Rotate entity inplace about x-axis, returns self (floating interface).

**Parameters** angle – rotation angle in radians

rotate_y *(angle: float)* → DXFGraphic
Rotate entity inplace about y-axis, returns self (floating interface).

**Parameters** angle – rotation angle in radians

rotate_z *(angle: float)* → DXFGraphic
Rotate entity inplace about z-axis, returns self (floating interface).

**Parameters** angle – rotation angle in radians

rotate_axis *(axis: Vec3, angle: float)* → DXFGraphic
Rotate entity inplace about vector axis, returns self (floating interface).

**Parameters**

- **axis** – rotation axis as tuple or Vec3
- **angle** – rotation angle in radians
Common graphical DXF attributes

DXFGraphic.dxf.layer
Layer name as string; default = '0'

DXFGraphic.dxf.linetype
Linetype as string, special names 'BYLAYER', 'BYBLOCK'; default value is 'BYLAYER'

DXFGraphic.dxf.color
AutoCAD Color Index (ACI), default = 256
  Constants defined in ezdxf.lldxf.const

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BYBLOCK</td>
</tr>
<tr>
<td>256</td>
<td>BYLAYER</td>
</tr>
<tr>
<td>257</td>
<td>BYOBJECT</td>
</tr>
</tbody>
</table>

DXFGraphic.dxf.lineweight
Line weight in mm times 100 (e.g. 0.13mm = 13). There are fixed valid lineweights which are accepted by AutoCAD, other values prevents AutoCAD from loading the DXF document, BricsCAD isn’t that picky. (requires DXF R2000)
  Constants defined in ezdxf.lldxf.const

<table>
<thead>
<tr>
<th></th>
<th>LINEWEIGHT_DEFAULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>LINEWEIGHT_BYLAYER</td>
</tr>
<tr>
<td>-2</td>
<td>LINEWEIGHT_BYBLOCK</td>
</tr>
<tr>
<td>-3</td>
<td>LINEWEIGHT_DEFAULT</td>
</tr>
</tbody>
</table>

Valid DXF lineweights stored in VALID_DXF_LINEWEIGHTS: 0, 5, 9, 13, 15, 18, 20, 25, 30, 35, 40, 50, 53, 60, 70, 80, 90, 100, 106, 120, 140, 158, 200, 211

DXFGraphic.dxf.ltscale
Line type scale as float; default = 1.0 (requires DXF R2000)

DXFGraphic.dxf.invisible
1 for invisible, 0 for visible; default = 0 (requires DXF R2000)

DXFGraphic.dxf.paperspace
0 for entity resides in modelspace or a block, 1 for paperspace, this attribute is set automatically by adding an entity to a layout (feature for experts); default = 0

DXFGraphic.dxf.extrusion
Extrusion direction as 3D vector; default = (0, 0, 1)

DXFGraphic.dxf.thickness
Entity thickness as float; default = 0.0 (requires DXF R2000)

DXFGraphic.dxf.true_color
True color value as int 0x00RRGGBB, use DXFGraphic.rgb to get/set true color values as (r, g, b) tuples. (requires DXF R2004)

DXFGraphic.dxf.color_name
Color name as string. (requires DXF R2004)

DXFGraphic.dxf.transparency
Transparency value as int, 0x000000TT 0x00 = 100% transparent / 0xFF = opaque, use DXFGraphic.transparency to get/set transparency as float value.
  (requires DXF R2004)
DXFGraphic.dxf.shadow_mode

<table>
<thead>
<tr>
<th></th>
<th>casts and receives shadows</th>
<th>1</th>
<th>casts shadows</th>
<th>2</th>
<th>receives shadows</th>
<th>3</th>
<th>ignores shadows</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

(requires DXF R2007)

Face3d

A 3DFACE (DXF Reference) is real 3D solid filled triangle or quadrilateral. Access vertices by name (entity.dxf.vtx0 = (1.7, 2.3)) or by index (entity[0] = (1.7, 2.3)).

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'3DFACE'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_3dface()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
</tbody>
</table>

**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.Face3d

```
Face3d because 3dface is not a valid Python class name.
```

dxf.vtx0
Location of 1. vertex (3D Point in WCS)

dxf.vtx1
Location of 2. vertex (3D Point in WCS)

dxf.vtx2
Location of 3. vertex (3D Point in WCS)

dxf.vtx3
Location of 4. vertex (3D Point in WCS)

dxf.invisible_edge
invisible edge flag (int, default=0)

<table>
<thead>
<tr>
<th></th>
<th>first edge is invisible</th>
<th>2</th>
<th>second edge is invisible</th>
<th>4</th>
<th>third edge is invisible</th>
<th>8</th>
<th>fourth edge is invisible</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td>4</td>
<td></td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Combine values by adding them, e.g. 1+4 = first and third edge is invisible.

transform (m: Matrix44) → Face3d

Transform the 3DFACE entity by transformation matrix m inplace.

wcs_vertices (close: bool=False) → List[Vec3]

Returns WCS vertices, if argument close is True, last vertex == first vertex. Does not return duplicated last vertex if represents a triangle.
Compatibility interface to SOLID and TRACE. The 3DFACE entity returns already WCS vertices.

**Solid3d**

3DSOLID (DXF Reference) created by an ACIS based geometry kernel provided by the Spatial Corp. ezdxf will never interpret ACIS source code, don’t ask me for this feature.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'3DSOLID'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_3dsolid()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.Solid3d

Same attributes and methods as parent class Body.

dxf.history_handle
Handle to history object.

**Arc**

ARC (DXF Reference) center at location dxf.center and radius of dxf.radius from dxf.start_angle to dxf.end_angle. ARC goes always from dxf.start_angle to dxf.end_angle in counter clockwise orientation around the dxf.extrusion vector, which is (0, 0, 1) by default and the usual case for 2D arcs.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'ARC'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_arc()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
</tbody>
</table>

**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.Arc

dxf.center
Center point of arc (2D/3D Point in OCS)

dxf.radius
Radius of arc (float)

dxf.start_angle
Start angle in degrees (float)

dxf.end_angle
End angle in degrees (float)
**start_point**
Returns the start point of the arc in WCS, takes OCS into account.

**end_point**
Returns the end point of the arc in WCS, takes OCS into account.

**angles** (num: int) \(\rightarrow\) Iterable[float]
Returns num angles from start- to end angle in degrees in counter clockwise order.
All angles are normalized in the range from \([0, 360)\).

**flattening** (sagitta: float) \(\rightarrow\) Iterable[Vertex]
Approximate the arc by vertices in WCS, argument segment is the max. distance from the center of an arc segment to the center of its chord. Yields Vec2 objects for 2D arcs and Vec3 objects for 3D arcs.
New in version 0.15.

**transform** (m: Matrix44) \(\rightarrow\) Arc
Transform ARC entity by transformation matrix m inplace.
Raises NonUniformScalingError() for non uniform scaling.

**to_ellipse** (replace=True) \(\rightarrow\) Ellipse
Convert CIRCLE/ARC to an Ellipse entity.
Adds the new ELLIPSE entity to the entity database and to the same layout as the source entity.

**Parameters replace** – replace (delete) source entity by ELLIPSE entity if True

**to_spline** (replace=True) \(\rightarrow\) Spline
Convert CIRCLE/ARC to a Spline entity.
Adds the new SPLINE entity to the entity database and to the same layout as the source entity.

**Parameters replace** – replace (delete) source entity by SPLINE entity if True

**construction_tool** () \(\rightarrow\) ConstructionArc
Returns 2D construction tool ezdxf.math.ConstructionArc, ignoring the extrusion vector.

**apply_construction_tool** (arc: ConstructionArc) \(\rightarrow\) Arc
Set ARC data from construction tool ezdxf.math.ConstructionArc, will not change the extrusion vector.

---

**Body**

BODY (DXF Reference) created by an ACIS based geometry kernel provided by the Spatial Corp. ezdxf will never interpret ACIS source code, don’t ask me for this feature.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'BODY'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_body()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!
class ezdxf.entities.Body

dxf.version
    Modeler format version number, default value is 1.

dxf.flags
    Require DXF R2013.

dxf.uid
    Require DXF R2013.

acis_data
    Get/Set ACIS text data as list of strings for DXF R2000 to R2010 and binary encoded ACIS data for DXF R2013 and later as list of bytes.

has_binary_data
    Returns True if ACIS data is of type List[bytes], False if data is of type List[str].

tostring() → str
    Returns ACIS data as one string for DXF R2000 to R2010.

tobytes() → bytes
    Returns ACIS data as joined bytes for DXF R2013 and later.

set_text(text: str, sep: str = \'\n\') → None
    Set ACIS data from one string.

Circle

CIRCLE (DXF Reference) center at location dxf.center and radius of dxf.radius.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'CIRCLE'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_circle()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
</tbody>
</table>

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.Circle

dxf.center
    Center point of circle (2D/3D Point in OCS)

dxf.radius
    Radius of circle (float)

vertices(angle:Iterable[float]) → Iterable[Vec3]
    Yields vertices of the circle for iterable angles in WCS.

    Parameters angles – iterable of angles in OCS as degrees, angle goes counter clockwise around the extrusion vector, ocs x-axis = 0 deg.

flattening(sagitta: float) → Iterable[Vec3]
    Approximate the circle by vertices in WCS, argument segment is the max. distance from the center of an arc segment to the center of its chord. Returns a closed polygon: start vertex == end vertex!
Yields always Vec3 objects.

New in version 0.15.

**transform** \((m: \text{Matrix44}) \rightarrow \text{Circle}\)

Transform the CIRCLE entity by transformation matrix \(m\) inplace.

Raises NonUniformScalingError() for non uniform scaling.

**translate** \((dx: \text{float}, dy: \text{float}, dz: \text{float}) \rightarrow \text{Circle}\)

Optimized CIRCLE/ARC translation about \(dx\) in x-axis, \(dy\) in y-axis and \(dz\) in z-axis, returns self (floating interface).

**to_ellipse** \((\text{replace=True}) \rightarrow \text{Ellipse}\)

Convert CIRCLE/ARC to an Ellipse entity.

Adds the new ELLIPSE entity to the entity database and to the same layout as the source entity.

**Parameters replace** – replace (delete) source entity by ELLIPSE entity if True

**to_spline** \((\text{replace=True}) \rightarrow \text{Spline}\)

Convert CIRCLE/ARC to a Spline entity.

Adds the new SPLINE entity to the entity database and to the same layout as the source entity.

**Parameters replace** – replace (delete) source entity by SPLINE entity if True

### Dimension

The DIMENSION entity (DXF Reference) represents several types of dimensions in many orientations and alignments. The basic types of dimensioning are linear, radial, angular, ordinate, and arc length.

For more information about dimensions see the online help from AutoDesk: About the Types of Dimensions

**Important:** The DIMENSION entity is reused to create dimensional constraints, such entities do not have an associated geometrical block nor a dimension type group code (2) and reside on layer *ADSK_CONSTRAINTS*. Use property Dimension.is_dimensional_constraint to check for this objects. Dimensional constraints are not documented in the DXF reference and not supported by ezdxf.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'DIMENSION'</td>
</tr>
<tr>
<td>factory function</td>
<td>see table below</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
</tbody>
</table>

#### Factory Functions

<table>
<thead>
<tr>
<th>Linear and Rotated Dimension (DXF)</th>
<th>add_linear_dim()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aligned Dimension (DXF)</td>
<td>add_aligned_dim()</td>
</tr>
<tr>
<td>Angular Dimension (DXF)</td>
<td>add_angular_dim() (not implemented)</td>
</tr>
<tr>
<td>Angular 3P Dimension (DXF)</td>
<td>add_angular_3p_dim() (not implemented)</td>
</tr>
<tr>
<td>Diameter Dimension (DXF)</td>
<td>add_diameter_dim()</td>
</tr>
<tr>
<td>Radius Dimension (DXF)</td>
<td>add_radius_dim()</td>
</tr>
<tr>
<td>Ordinate Dimension (DXF)</td>
<td>add_ordinate_dim() (not implemented)</td>
</tr>
</tbody>
</table>
**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!

**class ezdxf.entities.Dimension**

There is only one `Dimension` class to represent all different dimension types.

```python
dxf.version

Version number: 0 = R2010. (int, DXF R2010)
```

```python
dxf.geometry

Name of the BLOCK that contains the entities that make up the dimension picture.

For AutoCAD this graphical representation is mandatory, else AutoCAD will not open the DXF drawing.

BricsCAD will render the DIMENSION entity by itself, if the graphical representation is not present, but uses the BLOCK instead of rendering, if it is present.
```

```python
dxf.dimstyle

Dimension style (`DimStyle`) name as string.
```

```python
dxf.dimtype

Values 0-6 are integer values that represent the dimension type. Values 32, 64, and 128 are bit values, which are added to the integer values.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Linear and Rotated Dimension (DXF)</td>
</tr>
<tr>
<td>1</td>
<td>Aligned Dimension (DXF)</td>
</tr>
<tr>
<td>2</td>
<td>Angular Dimension (DXF)</td>
</tr>
<tr>
<td>3</td>
<td>Diameter Dimension (DXF)</td>
</tr>
<tr>
<td>4</td>
<td>Radius Dimension (DXF)</td>
</tr>
<tr>
<td>5</td>
<td>Angular 3P Dimension (DXF)</td>
</tr>
<tr>
<td>6</td>
<td>Ordinate Dimension (DXF)</td>
</tr>
<tr>
<td>8</td>
<td>Subclass <code>ezdxf.entities.ArcDimension</code> introduced in DXF R2004</td>
</tr>
<tr>
<td>32</td>
<td>Indicates that graphical representation <code>geometry</code> is referenced by this dimension only. (always set in DXF R13 and later)</td>
</tr>
<tr>
<td>64</td>
<td>Ordinate type. This is a bit value (bit 7) used only with integer value 6. If set, ordinate is <code>X-type</code>; if not set, ordinate is <code>Y-type</code></td>
</tr>
<tr>
<td>128</td>
<td>This is a bit value (bit 8) added to the other <code>dimtype</code> values if the dimension text has been positioned at a user-defined location rather than at the default location</td>
</tr>
</tbody>
</table>
```

```python
dxf.defpoint

Definition point for all dimension types. (3D Point in `WCS`)  

Linear and rotated dimension: `dxf.defpoint` specifies the dimension line location.  

Arc and angular dimension: `dxf.defpoint` and `dxf.defpoint4` specify the endpoints of the line used to determine the second extension line.
```

```python
dxf.defpoint2

Definition point for linear and angular dimensions. (3D Point in `WCS`)  

Linear and rotated dimension: The `dxf.defpoint2` specifies the start point of the first extension line.  

Arc and angular dimension: The `dxf.defpoint2` and `dxf.defpoint3` specify the endpoints of the line used to determine the first extension line.
```

```python
dxf.defpoint3

Definition point for linear and angular dimensions. (3D Point in `WCS`)  

Linear and rotated dimension: The `dxf.defpoint3` specifies the start point of the second extension line.
Arc and angular dimension: The `dxf.defpoint2` and `dxf.defpoint3` specify the endpoints of the line used to determine the first extension line.

`dxf.defpoint4`  
Definition point for diameter, radius, and angular dimensions. (3D Point in *WCS*)

Arc and angular dimension: `dxf.defpoint` and `dxf.defpoint4` specify the endpoints of the line used to determine the second extension line.

`dxf.defpoint5`  
Point defining dimension arc for angular dimensions, specifies the location of the dimension line arc. (3D Point in *OCS*)

`dxf.angle`  
Angle of linear and rotated dimensions in degrees. (float)

`dxf.leader_length`  
Leader length for radius and diameter dimensions. (float)

`dxf.text_midpoint`  
Middle point of dimension text. (3D Point in *OCS*)

`dxf.insert`  
Insertion point for clones of a linear dimensions—Baseline and Continue. (3D Point in *OCS*)

This value is used by CAD application (Baseline and Continue) and ignored by `ezdxf`.

`dxf.attachment_point`  
Text attachment point (int, DXF R2000), default value is 5.

<table>
<thead>
<tr>
<th>Value</th>
<th>Attachment Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Top left</td>
</tr>
<tr>
<td>2</td>
<td>Top center</td>
</tr>
<tr>
<td>3</td>
<td>Top right</td>
</tr>
<tr>
<td>4</td>
<td>Middle left</td>
</tr>
<tr>
<td>5</td>
<td>Middle center</td>
</tr>
<tr>
<td>6</td>
<td>Middle right</td>
</tr>
<tr>
<td>7</td>
<td>Bottom left</td>
</tr>
<tr>
<td>8</td>
<td>Bottom center</td>
</tr>
<tr>
<td>9</td>
<td>Bottom right</td>
</tr>
</tbody>
</table>

`dxf.line_spacing_style`  
Dimension text line-spacing style (int, DXF R2000), default value is 1.

<table>
<thead>
<tr>
<th>Value</th>
<th>Line-Spacing Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>At least (taller characters will override)</td>
</tr>
<tr>
<td>2</td>
<td>Exact (taller characters will not override)</td>
</tr>
</tbody>
</table>

`dxf.line_spacing_factor`  
Dimension text-line spacing factor. (float, DXF R2000)

Percentage of default (3-on-5) line spacing to be applied. Valid values range from 0.25 to 4.00.

`dxf.actual_measurement`  
Actual measurement (float, DXF R2000), this is an optional attribute and often not present. (read-only value)

`dxf.text`  
Dimension text explicitly entered by the user (str), default value is an empty string.
If empty string or '<>', the dimension measurement is drawn as the text, if ' ' (one blank space), the text is suppressed. Anything else is drawn as the text.

\texttt{dxf.oblique\_angle}  
Linear dimension types with an oblique angle have an optional \texttt{dxf.oblique\_angle}.  
When added to the rotation \texttt{dxf.angle} of the linear dimension, it gives the angle of the extension lines.

\texttt{dxf.text\_rotation}  
Defines the rotation angle of the dimension text away from its default orientation (the direction of the dimension line). (float)

\texttt{dxf.horizontal\_direction}  
Indicates the horizontal direction for the dimension entity (float).  
This attribute determines the orientation of dimension text and lines for horizontal, vertical, and rotated linear dimensions. This value is the negative of the angle in the OCS xy-plane between the dimension line and the OCS x-axis.

\texttt{get\_dim\_style}() \rightarrow \texttt{DimStyle}  
Returns the associated \texttt{DimStyle} entity.

\texttt{get\_geometry\_block}() \rightarrow \texttt{Optional[BlockLayout]}  
Returns \texttt{BlockLayout} of associated anonymous dimension block, which contains the entities that make up the dimension picture. Returns \texttt{None} if block name is not set or the BLOCK itself does not exist.

\texttt{get\_measurement}() \rightarrow \texttt{Union[float, ezdxf.math._vector.Vec3]}  
Returns the actual dimension measurement in \textit{WCS} units, no scaling applied for linear dimensions. Returns angle in degrees for angular dimension from 2 lines and angular dimension from 3 points. Returns vector from origin to feature location for ordinate dimensions.

\texttt{override}() \rightarrow \texttt{DimStyleOverride}  
Returns the \texttt{DimStyleOverride} object.

\texttt{render}()  
Render graphical representation as anonymous block.

\texttt{transform}(m: Matrix44) \rightarrow \texttt{Dimension}  
Transform the \texttt{DIMENSION} entity by transformation matrix \textit{m} inplace.  
Raises \texttt{NonUniformScalingError()} for non uniform scaling.

\texttt{virtual\_entities}() \rightarrow \texttt{Iterable[DXFGraphic]}  
Yields ‘virtual’ parts of \texttt{DIMENSION} as basic DXF entities like LINE, ARC or TEXT.  
This entities are located at the original positions, but are not stored in the entity database, have no handle and are not assigned to any layout.

\texttt{explode}(target\_layout: BaseLayout = None) \rightarrow \texttt{EntityQuery}  
Explode parts of \texttt{DIMENSION} as basic DXF entities like LINE, ARC or TEXT into target layout, if target layout is \texttt{None}, the target layout is the layout of the \texttt{DIMENSION}.  
Returns an \texttt{EntityQuery} container with all DXF primitives.

\texttt{DimStyleOverride}  
All of the \texttt{DimStyle} attributes can be overridden for each \texttt{Dimension} entity individually.
The `DimStyleOverride` class manages all the complex dependencies between `DimStyle` and `Dimension`, the different features of all DXF versions and the rendering process to create the `Dimension` picture as BLOCK, which is required for AutoCAD.

```python
class ezdxf.entities.DimStyleOverride

dimension
    Base `Dimension` entity.

dimstyle
    By `dimension` referenced `DimStyle` entity.

dimstyle_attribs
    Contains all overridden attributes of `dimension`, as a dict with `DimStyle` attribute names as keys.

__getitem__(key: str) → Any
    Returns `DIMSTYLE` attribute `key`, see also `get()`.

__setitem__(key: str, value: Any) → None
    Set `DIMSTYLE` attribute `key` in `dimstyle_attribs`.

__delitem__(key: str) → None
    Deletes `DIMSTYLE` attribute `key` from `dimstyle_attribs`, ignores `KeyErrors` silently.

get(attribute: str, default: Any = None) → Any
    Returns `DIMSTYLE` attribute from override dict `dimstyle_attribs` or base `DimStyle`.
    Returns default value for attributes not supported by DXF R12. This is a hack to use the same algorithm to render DXF R2000 and DXF R12 DIMENSION entities. But the DXF R2000 attributes are not stored in the DXF R12 file! Does not catch invalid attributes names! Look into debug log for ignored `DIMSTYLE` attributes.

pop(attribute: str, default: Any = None) → Any
    Returns `DIMSTYLE` attribute from override dict `dimstyle_attribs` and removes this `attribute` from override dict.

update(attribs: dict) → None
    Update override dict `dimstyle_attribs`.

commit() → None
    Writes overridden `DIMSTYLE` attributes into `ACAD:DSTYLE` section of XDATA of the `DIMENSION` entity.

get_arrow_names() → Tuple[str, str]
    Get arrow names as strings like 'ARCHTICK'.
    **Returns** tuple of [dimblk1, dimblk2]

set_arrows(blk: str = None, blk1: str = None, blk2: str = None, ldrblk: str = None, size: float = None) → None
    Set arrows or user defined blocks and disable oblique stroke as tick.
    **Parameters**
    - `blk` – defines both arrows at once as name str or user defined block
    - `blk1` – defines left arrow as name str or as user defined block
    - `blk2` – defines right arrow as name str or as user defined block
```
• `ldrblk` – defines leader arrow as name str or as user defined block
• `size` – arrow size in drawing units

**set_tick** *(size: float = 1) → None*
Use oblique stroke as tick, disables arrows.

Parameters *size* – arrow size in daring units

**set_text_align** *(halign: str = None, valign: str = None, vshift: float = None) → None*
Set measurement text alignment, *halign* defines the horizontal alignment, *valign* defines the vertical alignment, *above1* and *above2* means above extension line 1 or 2 and aligned with extension line.

Parameters

• *halign* – *left*, *right*, *center*, *above1*, *above2*, requires DXF R2000+
• *valign* – *above*, *center*, *below*
• *vshift* – vertical text shift, if *valign* is *center*; >0 shift upward, <0 shift downwards

**set_tolerance** *(upper: float, lower: float = None, hfactor: float = None, align: str = None, dec: int = None, leading_zeros: bool = None, trailing_zeros: bool = None) → None*
Set tolerance text format, upper and lower value, text height factor, number of decimal places or leading and trailing zero suppression.

Parameters

• *upper* – upper tolerance value
• *lower* – lower tolerance value, if None same as upper
• *hfactor* – tolerance text height factor in relation to the dimension text height
• *align* – tolerance text alignment “TOP”, “MIDDLE”, “BOTTOM”
• *dec* – Sets the number of decimal places displayed
• *leading_zeros* – suppress leading zeros for decimal dimensions if False
• *trailing_zeros* – suppress trailing zeros for decimal dimensions if False

**set_limits** *(upper: float, lower: float, hfactor: float = None, dec: int = None, leading_zeros: bool = None, trailing_zeros: bool = None) → None*
Set limits text format, upper and lower limit values, text height factor, number of decimal places or leading and trailing zero suppression.

Parameters

• *upper* – upper limit value added to measurement value
• *lower* – lower limit value subtracted from measurement value
• *hfactor* – limit text height factor in relation to the dimension text height
• *dec* – Sets the number of decimal places displayed, required DXF R2000+
• *leading_zeros* – suppress leading zeros for decimal dimensions if False, required DXF R2000+
• *trailing_zeros* – suppress trailing zeros for decimal dimensions if False, required DXF R2000+

**set_text_format** *(prefix: str = ",* postfix: str = ",* rnd: float = None, dec: int = None, sep: str = None, leading_zeros: bool = None, trailing_zeros: bool = None) → None*
Set dimension text format, like prefix and postfix string, rounding rule and number of decimal places.

Parameters
• **prefix** – dimension text prefix text as string
• **postfix** – dimension text postfix text as string
• **rnd** – Rounds all dimensioning distances to the specified value, for instance, if DIMRND is set to 0.25, all distances round to the nearest 0.25 unit. If you set DIMRND to 1.0, all distances round to the nearest integer.
• **dec** – Sets the number of decimal places displayed for the primary units of a dimension. Requires DXF R2000+
• **sep** – “.” or “,” as decimal separator
• **leading_zeros** – suppress leading zeros for decimal dimensions if False
• **trailing_zeros** – suppress trailing zeros for decimal dimensions if False

```python
set_dimline_format(color: int = None, linetype: str = None, lineweight: int = None, extension: float = None, disable1: bool = None, disable2: bool = None)
```
Set dimension line properties

**Parameters**
• **color** – color index
• **linetype** – linetype as string
• **lineweight** – line weight as int, 13 = 0.13mm, 200 = 2.00mm
• **extension** – extension length
• **disable1** – True to suppress first part of dimension line
• **disable2** – True to suppress second part of dimension line

```python
set_extline_format(color: int = None, lineweight: int = None, extension: float = None, offset: float = None, fixed_length: float = None)
```
Set common extension line attributes.

**Parameters**
• **color** – color index
• **lineweight** – line weight as int, 13 = 0.13mm, 200 = 2.00mm
• **extension** – extension length above dimension line
• **offset** – offset from measurement point
• **fixed_length** – set fixed length extension line, length below the dimension line

```python
set_extline1(linetype: str = None, disable=False)
```
Set extension line 1 attributes.

**Parameters**
• **linetype** – linetype for extension line 1
• **disable** – disable extension line 1 if True

```python
set_extline2(linetype: str = None, disable=False)
```
Set extension line 2 attributes.

**Parameters**
• **linetype** – linetype for extension line 2
• **disable** – disable extension line 2 if True
**set_text** *(text: str = '<>') → None*

Set dimension text.

- `text = " "` to suppress dimension text
- `text = ""` or `"<>"` to use measured distance as dimension text
- else use “text” literally

**shift_text** *(dh: float, dv: float) → None*

Set relative text movement, implemented as user location override without leader.

**Parameters**

- `dh` – shift text in text direction
- `dv` – shift text perpendicular to text direction

**set_location** *(location: Vertex, leader=False, relative=False) → None*

Set text location by user, special version for linear dimensions, behaves for other dimension types like `user_location_override()`.

**Parameters**

- `location` – user defined text location (Vertex)
- `leader` – create leader from text to dimension line
- `relative` – `location` is relative to default location.

**user_location_override** *(location: Vertex) → None*

Set text location by user, `location` is relative to the origin of the UCS defined in the `render()` method or WCS if the `ucs` argument is `None`.

**render** *(ucs: UCS = None, discard=False) → BaseDimensionRenderer*

Initiate dimension line rendering process and also writes overridden dimension style attributes into the DSTYLE XDATA section.

For a friendly CAD applications like BricsCAD you can discard the dimension line rendering, because it is done automatically by BricsCAD, if no dimension rendering BLOCK is available and it is likely to get better results as by ezdxf.

AutoCAD does not render DIMENSION entities automatically, so I rate AutoCAD as an unfriendly CAD application.

**Parameters**

- `ucs` – user coordinate system
- `discard` – discard rendering done by ezdxf (works with BricsCAD, but not tolerated by AutoCAD)

**Returns** Rendering object used to render the DIMENSION entity for analytics

**Return type** BaseDimensionRenderer

**ArcDimension**

The ARC_DIMENSION entity was introduced in DXF R2004 and is **not** documented in the DXF reference.
ezdxf Documentation, Release 0.16.2

### Arc Dimension

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'ARC_DIMENSION'</td>
</tr>
<tr>
<td>factory function</td>
<td>add_arc_dim() (not implemented)</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2004 ('AC1018')</td>
</tr>
</tbody>
</table>

**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!

```python
class ezdxf.entities.ArcDimension
    dxf.ext_line1_point
    dxf.ext_line2_point
    dxf.arc_center
    dxf.start_angle
    dxf.end_angle
    dxf.is_partial
    dxf.has_leader
    dxf.leader_point1
    dxf.leader_point2
    dimtype
        Returns always 8.
```

### Ellipse

ELLIPSE (DXF Reference) with center point at location `dxf.center` and a major axis `dxf.major_axis` as vector. `dxf.ratio` is the ratio of minor axis to major axis. `dxf.start_param` and `dxf.end_param` defines the starting- and the end point of the ellipse, a full ellipse goes from 0 to 2*pi. The ellipse goes from starting- to end param in counter clockwise direction.

`dxf.extrusion` is supported, but does not establish an OCS, but creates an 3D entity by extruding the base ellipse in direction of the `dxf.extrusion` vector.

```python
class ezdxf.entities.Ellipse
    dxf.center
        Center point of circle (2D/3D Point in WCS)
    dxf.major_axis
        Endpoint of major axis, relative to the dxf.center (Vec3), default value is (1, 0, 0).
```

---

6.8. Reference 229
dxf.ratio
Ratio of minor axis to major axis (float), has to be in range from 0.000001 to 1, default value is 1.

dxf.start_param
Start parameter (float), default value is 0.

dxf.end_param
End parameter (float), default value is 2*pi.

start_point
Returns the start point of the ellipse in WCS.

df.end_point
Returns the end point of the ellipse in WCS.

minor_axis
Returns the minor axis of the ellipse as Vec3 in WCS.

constrution_tool() → ConstructionEllipse
Returns construction tool ezdxl.math.ConstructionEllipse.

apply_construction_tool(e: ConstructionEllipse) → Ellipse
Set ELLIPSE data from construction tool ezdxl.math.ConstructionEllipse.

vertices (params:Iterable[float]) → Iterable[Vec3]
Yields vertices on ellipse for iterable params in WCS.

Parameters params – param values in the range from 0 to 2*pi in radians, param goes
counter clockwise around the extrusion vector, major_axis = local x-axis = 0 rad.

flattening (distance: float, segments: int = 8) → Iterable[Vec3]
Adaptive recursive flattening. The argument segments is the minimum count of approximation segments, if
the distance from the center of the approximation segment to the curve is bigger than distance the segment
will be subdivided. Returns a closed polygon for a full ellipse: start vertex == end vertex.

Parameters

• distance – maximum distance from the projected curve point onto the segment chord.

• segments – minimum segment count

New in version 0.15.

params (num: int) → Iterable[float]
Returns num params from start- to end param in counter clockwise order.
All params are normalized in the range from [0, 2pi).

transform (m: Matrix44) → Ellipse
Transform the ELLIPSE entity by transformation matrix m inplace.

translate (dx: float, dy: float, dz: float) → Ellipse
Optimized ELLIPSE translation about dx in x-axis, dy in y-axis and dz in z-axis, returns self (floating
interface).

to_spline (replace=True) → Spline
Convert ELLIPSE to a Spline entity.

Parameters replace – replace (delete) source entity by SPLINE entity if True

classmethod from_arc (entity: DXFGraphic) → Ellipse
Create a new ELLIPSE entity from ARC or CIRCLE entity.
The new SPLINE entity has no owner, no handle, is not stored in the entity database nor assigned to any layout!

Hatch

The HATCH entity (DXF Reference) fills an enclosed area defined by one or more boundary paths with a hatch pattern, solid fill, or gradient fill.

All points in OCS as (x, y) tuples (Hatch.dxf.elevation is the z-axis value).

There are two different hatch pattern default scaling, depending on the HEADER variable $MEASUREMENT, one for ISO measurement (m, cm, mm, ...) and one for imperial measurement (in, ft, yd, ...).

Starting with ezdxf v0.15 the default scaling for predefined hatch pattern will be chosen according this measurement setting in the HEADER section, this replicates the behavior of BricsCAD and other CAD applications. ezdxf uses the ISO pattern definitions as a base line and scales this pattern down by factor 1/25.6 for imperial measurement usage. The pattern scaling is independent from the drawing units of the document defined by the HEADER variable $INSUNITS.

Prior to ezdxf v0.15 the default scaling was always the ISO measurement scaling, no matter which value $MEASUREMENT had.

See also:

Tutorial for Hatch and DXF Units

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'HATCH'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_hatch()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

Boundary paths helper classes

Path manager: BoundaryPaths

- PolylinePath
- EdgePath
  - LineEdge
  - ArcEdge
  - EllipseEdge
  - SplineEdge

Pattern and gradient helper classes

- Pattern
- PatternLine
- Gradient
class ezdxf.entities.Hatch
dx.

*pattern_name*
Pattern name as string

*solid_fill*

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>solid fill, better use: <code>Hatch.set_solid_fill()</code></td>
</tr>
<tr>
<td>0</td>
<td>pattern fill, better use: <code>Hatch.set_pattern_fill()</code></td>
</tr>
</tbody>
</table>

*associative*

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>associative hatch</td>
</tr>
<tr>
<td>0</td>
<td>not associative hatch</td>
</tr>
</tbody>
</table>

Associations not handled by ezdxf, you have to set the handles to the associated DXF entities by yourself.

*dx.

*hatch_style*

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>normal</td>
</tr>
<tr>
<td>1</td>
<td>outer</td>
</tr>
<tr>
<td>2</td>
<td>ignore</td>
</tr>
</tbody>
</table>

(search AutoCAD help for more information)

*dx.

*pattern_type*

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>user</td>
</tr>
<tr>
<td>1</td>
<td>predefined</td>
</tr>
<tr>
<td>2</td>
<td>custom</td>
</tr>
</tbody>
</table>

*dx.

*pattern_angle*
Actual pattern angle in degrees (float). Changing this value does not rotate the pattern, use `set_pattern_angle()` for this task.

*dx.

*pattern_scale*
Actual pattern scaling factor (float). Changing this value does not scale the pattern use `set_pattern_scale()` for this task.

*dx.

*pattern_double*
1 = double pattern size else 0. (int)

*dx.

*n_seed_points*
Count of seed points (better user: `get_seed_points()`)

*dx.

*elevation*
Z value represents the elevation height of the OCS. (float)

*paths*
`BoundaryPaths` object.

*pattern*
`Pattern` object.
**gradient**

*Gradient* object.

**seeds**

List of (x, y) tuples.

**has_solid_fill**

True if hatch has a solid fill. (read only)

**has_pattern_fill**

True if hatch has a pattern fill. (read only)

**has_gradient_data**

True if hatch has a gradient fill. A hatch with gradient fill has also a solid fill. (read only)

**bgcolor**

Property background color as (r, g, b)-tuple, rgb values in the range [0, 255] (read/write/del)

Usage:

```python
color = hatch.bgcolor  # get background color as (r, g, b) tuple
hatch.bgcolor = (10, 20, 30)  # set background color
del hatch.bgcolor  # delete background color
```

**set_pattern_definition** (lines: Sequence[T_co], factor: float = 1, angle: float = 0) → None

Setup hatch pattern definition by a list of definition lines and a definition line is a 4-tuple [angle, base_point, offset, dash_length_items], the pattern definition should be designed for scaling factor 1 and angle 0.

- angle: line angle in degrees
- base_point: 2-tuple (x, y)
- offset: 2-tuple (dx, dy)
- dash_length_items: list of dash items (item > 0 is a line, item < 0 is a gap and item == 0.0 is a point)

**Parameters**

- lines – list of definition lines
- factor – pattern scaling factor
- angle – rotation angle in degrees

Changed in version 0.13: added angle argument

**set_pattern_scale** (scale: float) → None

Set scaling of pattern definition to scale.

Starts always from the original base scaling, set_pattern_scale(1) reset the pattern scaling to the original appearance as defined by the pattern designer, but only if the the pattern attribute `dxf.PATTERN_SCALE` represents the actual scaling, it is not possible to recreate the original pattern scaling from the pattern definition itself.

**Parameters**

- scale – pattern scaling factor

**set_pattern_angle** (angle: float) → None

Set rotation of pattern definition to angle in degrees.

Starts always from the original base rotation 0, set_pattern_angle(0) reset the pattern rotation to the original appearance as defined by the pattern designer, but only if the the pattern attribute `dxf.PATTERN_ANGLE` represents the actual rotation, it is not possible to recreate the original rotation from the pattern definition itself.
Parameters **angle** – rotation angle in degrees

**set_solid_fill** *(color: int = 7, style: int = 1, rgb: RGB = None)*

Set Hatch to solid fill mode and removes all gradient and pattern fill related data.

**Parameters**

- **color** – AutoCAD Color Index (ACI), (0 = BYBLOCK; 256 = BYLAYER)
- **style** – hatch style (0 = normal; 1 = outer; 2 = ignore)
- **rgb** – true color value as (r, g, b)-tuple - has higher priority than color. True color support requires DXF R2000.

**set_pattern_fill** *(name: str, color: int = 7, angle: float = 0.0, scale: float = 1.0, double: int = 0, style: int = 1, pattern_type: int = 1, definition=None)* → None

Set Hatch to pattern fill mode. Removes all gradient related data. The pattern definition should be designed for scaling factor 1. Predefined hatch pattern like “ANSI33” are scaled according to the HEADER variable $MEASUREMENT for ISO measurement (m, cm, . . . ), or imperial units (in, ft, . . . ), this replicates the behavior of BricsCAD.

**Parameters**

- **name** – pattern name as string
- **color** – pattern color as AutoCAD Color Index (ACI)
- **angle** – angle of pattern fill in degrees
- **scale** – pattern scaling as float
- **double** – double size flag
- **style** – hatch style (0 = normal; 1 = outer; 2 = ignore)
- **pattern_type** – pattern type (0 = user-defined; 1 = predefined; 2 = custom)
- **definition** – list of definition lines and a definition line is a 4-tuple [angle, base_point, offset, dash_length_items], see **set_pattern_definition()**

**set_gradient** *(color1: RGB = (0, 0, 0), color2: RGB = (255, 255, 255), rotation: float = 0.0, centered: float = 0.0, one_color: int = 0, tint: float = 0.0, name: str = 'LINEAR')* → None

Set Hatch to gradient fill mode and removes all pattern fill related data. Gradient support requires DXF DXF R2004. A gradient filled hatch is also a solid filled hatch.

Valid gradient type names are:

- 'LINEAR'
- 'CYLINDER'
- 'INVCYLINDER'
- 'SPHERICAL'
- 'INVSFERICAL'
- 'HEMISPHERICAL'
- 'INVHEMISPHERICAL'
- 'CURVED'
- 'INVCURVED'

**Parameters**


• **color1** – (r, g, b)-tuple for first color, rgb values as int in the range [0, 255]
• **color2** – (r, g, b)-tuple for second color, rgb values as int in the range [0, 255]
• **rotation** – rotation angle in degrees
• **centered** – determines whether the gradient is centered or not
• **one_color** – 1 for gradient from color1 to tinted color1
• **tint** – determines the tinted target color1 for a one color gradient. (valid range 0.0 to 1.0)
• **name** – name of gradient type, default “LINEAR”

```python
def set_seed_points(points: Iterable[Tuple[float, float]]) → None
    Set seed points, points is an iterable of (x, y)-tuples. I don’t know why there can be more than one seed point. All points in OCS (Hatch.dxf.elevation is the Z value).
```

```python
def transform(m: Matrix44) → Hatch
    Transform the HATCH entity by transformation matrix m inplace.
```

```python
def associate(path: Union[PolylinePath, EdgePath], entities: Iterable[DXFEntity])
    Set association from hatch boundary path to DXF geometry entities.
```

```python
def remove_association()
    Remove associated path elements.
```

### Boundary Paths

The hatch entity is build by different functional path types, this are filter flags for the **Hatch.dxf.hatch_style**:

- **EXTERNAL**: defines the outer boundary of the hatch
- **OUTERMOST**: defines the first tier of inner hatch boundaries
- **DEFAULT**: default boundary path

As you will learn in the next sections, these are more the recommended usage type for the flags, but the fill algorithm doesn’t care much about that, for instance an OUTERMOST path doesn’t have to be inside the EXTERNAL path.

### Island Detection

In general the island detection algorithm works always from outside to inside and alternates filled and unfilled areas. The area between then 1st and the 2nd boundary is filled, the area between the 2nd and the 3rd boundary is unfilled and so on. The different hatch styles defined by the **Hatch.dxf.hatch_style** attribute are created by filtering some boundary path types.

### Hatch Style

- **HATCH_STYLE_IGNORE**: Ignores all paths except the paths marked as EXTERNAL, if there are more than one path marked as EXTERNAL, they are filled in NESTED style. Creates no hatch if no path is marked as EXTERNAL.
• **HATCH_STYLE_OUTERMOST**: Ignores all paths marked as DEFAULT, remaining EXTERNAL and OUTERMOST paths are filled in NESTED style. Creates no hatch if no path is marked as EXTERNAL or OUTERMOST.

• **HATCH_STYLE_NESTED**: Use all existing paths.

**Hatch Boundary Helper Classes**

class ezdxf.entities.BoundaryPaths

Defines the borders of the hatch, a hatch can consist of more than one path.

paths

List of all boundary paths. Contains PolylinePath and EdgePath objects. (read/write)

e external_paths () -> Iterable[Union[PolylinePath, EdgePath]]

Iterable of external paths, could be empty.

outermost_paths () -> Iterable[Union[PolylinePath, EdgePath]]

Iterable of outermost paths, could be empty.

default_paths () -> Iterable[Union[PolylinePath, EdgePath]]

Iterable of default paths, could be empty.

rendering_paths (hatch_style: int) -> Iterable[Union[PolylinePath, EdgePath]]

Iterable of paths to process for rendering, filters unused boundary paths according to the given hatch style:

• NESTED: use all boundary paths
• OUTERMOST: use EXTERNAL and OUTERMOST boundary paths
• IGNORE: ignore all paths except EXTERNAL boundary paths

Yields paths in order of EXTERNAL, OUTERMOST and DEFAULT.

add_polyline_path (path_vertices, is_closed=1, flags=1) -> PolylinePath

Create and add a new PolylinePath object.

Parameters

• **path_vertices** – list of polyline vertices as (x, y) or (x, y, bulge)-tuples.
• **is_closed** – 1 for a closed polyline else 0
• **flags** – external(1) or outermost(16) or default (0)

add_edge_path (flags=1) -> EdgePath

Create and add a new EdgePath object.

Parameters **flags** – external(1) or outermost(16) or default (0)

polyline_to_edge_path (just_with_bulge=True) -> None

Convert polyline paths including bulge values to line- and arc edges.

Parameters **just_with_bulge** – convert only polyline paths including bulge values if True

arc_edges_to_ellipse_edges () -> None

Convert all arc edges to ellipse edges.

e llipse_edges_to_spline_edges (num: int = 32) -> None

Convert all ellipse edges to spline edges (approximation).

Parameters **num** – count of control points for a full ellipse, partial ellipses have proportional fewer control points but at least 3.
**spline_edges_to_line_edges** *(factor: int = 8) → None*

Convert all spline edges to line edges (approximation).

**Parameters**

- **factor** – count of approximation segments = count of control points x factor

**all_to_spline_edges** *(num: int = 64) → None*

Convert all bulge, arc and ellipse edges to spline edges (approximation).

**Parameters**

- **num** – count of control points for a full circle/ellipse, partial circles/ellipses have proportional fewer control points but at least 3.

**all_to_line_edges** *(num: int = 64, spline_factor: int = 8) → None*

Convert all bulge, arc and ellipse edges to spline edges and approximate this splines by line edges.

**Parameters**

- **num** – count of control points for a full circle/ellipse, partial circles/ellipses have proportional fewer control points but at least 3.
- **spline_factor** – count of spline approximation segments = count of control points x spline_factor

**clear** () → None

Remove all boundary paths.

**class ezdxfs.entities.PolylinePath**

A polyline as hatch boundary path.

**path_type_flags** *(bit coded flags)*

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>default</td>
</tr>
<tr>
<td>1</td>
<td>external</td>
</tr>
<tr>
<td>2</td>
<td>polyline, will be set by ezdxfs</td>
</tr>
<tr>
<td>16</td>
<td>outermost</td>
</tr>
</tbody>
</table>

My interpretation of the **path_type_flags**, see also Tutorial for Hatch:

- external - path is part of the hatch outer border
- outermost - path is completely inside of one or more external paths
- default - path is completely inside of one or more outermost paths

If there are troubles with AutoCAD, maybe the hatch entity has the Hatch.dxf.pixel_size attribute set - delete it del hatch.dxf.pixel_size and maybe the problem is solved. ezdxfs does not use the Hatch.dxf.pixel_size attribute, but it can occur in DXF files created by other applications.

**is_closed**

True if polyline path is closed.

**vertices**

List of path vertices as (x, y, bulge)-tuples. (read/write)

**source_boundary_objects**

List of handles of the associated DXF entities for associative hatches. There is no support for associative hatches by ezdxfs, you have to do it all by yourself. (read/write)

**set_vertices** *(vertices: Sequence[Sequence[float]], is_closed: bool = True) → None*

Set new vertices as new polyline path, a vertex has to be a (x, y) or a (x, y, bulge)-tuple.

**clear** () → None

Removes all vertices and all handles to associated DXF objects (source_boundary_objects).
class ezdxf.entities.EdgePath
Boundary path build by edges. There are four different edge types: LineEdge, ArcEdge, EllipseEdge of SplineEdge. Make sure there are no gaps between edges. AutoCAD in this regard is very picky. ezdxf performs no checks on gaps between the edges.

path_type_flags
(bit coded flags)

<table>
<thead>
<tr>
<th></th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>default</td>
</tr>
<tr>
<td>1</td>
<td>external</td>
</tr>
<tr>
<td>16</td>
<td>outermost</td>
</tr>
</tbody>
</table>

see PolylinePath.path_type_flags

edges
List of boundary edges of type LineEdge, ArcEdge, EllipseEdge of SplineEdge

source_boundary_objects
Required for associative hatches, list of handles to the associated DXF entities.

clear () → None
Delete all edges.

add_line (start, end) → LineEdge
Add a LineEdge from start to end.

Parameters
• start – start point of line, (x, y)-tuple
• end – end point of line, (x, y)-tuple

add_arc (center, radius=1., start_angle=0., end_angle=360., ccw:bool=True) → ArcEdge
Add an ArcEdge.

Parameters
• center – center point of arc, (x, y)-tuple
• radius – radius of circle
• start_angle – start angle of arc in degrees
• end_angle – end angle of arc in degrees
• ccw – True for counter clockwise False for clockwise orientation

add_ellipse (center, major_axis_vector=(1., 0.), minor_axis_length=1., start_angle=0., end_angle=360., ccw:bool=True) → EllipsePath
Add an EllipseEdge.

Parameters
• center – center point of ellipse, (x, y)-tuple
• major_axis – vector of major axis as (x, y)-tuple
• ratio – ratio of minor axis to major axis as float
• start_angle – start angle of arc in degrees
• end_angle – end angle of arc in degrees
• ccw – True for counter clockwise False for clockwise orientation
add_spline(fit_points=None, control_points=None, knot_values=None, weights=None, degree=3, rational=0, periodic=0) → SplinePath

Add a SplineEdge.

Parameters

- **fit_points** – points through which the spline must go, at least 3 fit points are required. list of (x, y)-tuples
- **control_points** – affects the shape of the spline, mandatory and AutoCAD crashes on invalid data. list of (x, y)-tuples
- **knot_values** – (knot vector) mandatory and AutoCAD crashes on invalid data. list of floats; ezdxf provides two tool functions to calculate valid knot values: ezdxf.math.uniform_knot_vector(), ezdxf.math.open_uniform_knot_vector() (default if None)
- **weights** – weight of control point, not mandatory, list of floats.
- **degree** – degree of spline (int)
- **periodic** – 1 for periodic spline, 0 for none periodic spline
- **start_tangent** – start_tangent as 2d vector, optional
- **end_tangent** – end_tangent as 2d vector, optional

**Warning:** Unlike for the spline entity AutoCAD does not calculate the necessary knot_values for the spline edge itself. On the contrary, if the knot_values in the spline edge are missing or invalid AutoCAD crashes.

class ezdxf.entities.LineEdge
Straight boundary edge.

start
Start point as (x, y)-tuple. (read/write)

der
End point as (x, y)-tuple. (read/write)

class ezdxf.entities.ArcEdge
Arc as boundary edge.

center
Center point of arc as (x, y)-tuple. (read/write)

radius
Arc radius as float. (read/write)

start_angle
Arc start angle in degrees. (read/write)

dir
Arc end angle in degrees. (read/write)

ccw
True for counter clockwise arc else False. (read/write)

class ezdxf.entities.EllipseEdge
Elliptic arc as boundary edge.

6.8. Reference
**major_axis_vector**
Ellipse major axis vector as (x, y)-tuple. (read/write)

**minor_axis_length**
Ellipse minor axis length as float. (read/write)

**radius**
Ellipse radius as float. (read/write)

**start_angle**
Ellipse start angle in degrees. (read/write)

**end_angle**
Ellipse end angle in degrees. (read/write)

**ccw**
True for counter clockwise ellipse else False. (read/write)

**class ezdxf.entities.SplineEdge**
Spline as boundary edge.

**degree**
Spline degree as int. (read/write)

**rational**
1 for rational spline else 0. (read/write)

**periodic**
1 for periodic spline else 0. (read/write)

**knot_values**
List of knot values as floats. (read/write)

**control_points**
List of control points as (x, y)-tuples. (read/write)

**fit_points**
List of fit points as (x, y)-tuples. (read/write)

**weights**
List of weights (of control points) as floats. (read/write)

**start_tangent**
Spline start tangent (vector) as (x, y)-tuple. (read/write)

**end_tangent**
Spline end tangent (vector) as (x, y)-tuple. (read/write)

**Hatch Pattern Definition Helper Classes**

**class ezdxf.entities.Pattern**

**lines**
List of pattern definition lines (read/write). see PatternLine

**add_line**(angle: float = 0, base_point: Vertex = (0, 0), offset: Vertex = (0, 0), dash_length_items: Iterable[float] = None) → None
Create a new pattern definition line and add the line to the Pattern.lines attribute.

**clear**() → None
Delete all pattern definition lines.
scale \((factor: float = 1, angle: float = 0) \rightarrow None\)
Scale and rotate pattern.
Be careful, this changes the base pattern definition, maybe better use `Hatch.set_pattern_scale()` or `Hatch.set_pattern_angle()`.

Parameters
- **factor** – scaling factor
- **angle** – rotation angle in degrees

class `ezdxftype.entities.PatternLine`
Represents a pattern definition line, use factory function `Pattern.add_line()` to create new pattern definition lines.

angle
Line angle in degrees. (read/write)

base_point
Base point as \((x, y)\)-tuple. (read/write)

offset
Offset as \((x, y)\)-tuple. (read/write)

dash_length_items
List of dash length items (item > 0 is line, < 0 is gap, 0.0 = dot). (read/write)

---

**Hatch Gradient Fill Helper Classes**

class `ezdxftype.entities.Gradient`

color1
First rgb color as \((r, g, b)\)-tuple, rgb values in range 0 to 255. (read/write)

color2
Second rgb color as \((r, g, b)\)-tuple, rgb values in range 0 to 255. (read/write)

one_color
If `one_color` is 1 - the hatch is filled with a smooth transition between `color1` and a specified `tint` of `color1`. (read/write)

rotation
Gradient rotation in degrees. (read/write)

centered
Specifies a symmetrical gradient configuration. If this option is not selected, the gradient fill is shifted up and to the left, creating the illusion of a light source to the left of the object. (read/write)

tint
Specifies the tint \((color1 mixed with white)\) of a color to be used for a gradient fill of one color. (read/write)

See also:

* Tutorial for Hatch Pattern Definition*
Image

Add a raster IMAGE (DXF Reference) to the DXF file, the file itself is not embedded into the DXF file, it is always a separated file. The IMAGE entity is like a block reference, you can use it multiple times to add the image on different locations with different scales and rotations. But therefore you need a also a IMAGEDEF entity, see ImageDef. ezdxf creates only images in the xy-plan, you can place images in the 3D space too, but then you have to set the Image.dxf.u_pixel and the Image.dxf.v_pixel vectors by yourself.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'IMAGE'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_image()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.Image

dxf.insert
Insertion point, lower left corner of the image (3D Point in WCS).

dxf.u_pixel
U-vector of a single pixel (points along the visual bottom of the image, starting at the insertion point) as (x, y, z) tuple

dxf.v_pixel
V-vector of a single pixel (points along the visual left side of the image, starting at the insertion point) as (x, y, z) tuple

dxf.image_size
Image size in pixels as (x, y) tuple

dxf.image_def_handle
Handle to the image definition entity, see ImageDef

dxf.flags

<table>
<thead>
<tr>
<th>Image.dxf.flags</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image.SHOW_IMAGE</td>
<td>1</td>
<td>Show image</td>
</tr>
<tr>
<td>Image.SHOW_WHEN_NOT_ALIGNED</td>
<td>2</td>
<td>Show image when not aligned with screen</td>
</tr>
<tr>
<td>Image.USE_CLIPPING_BOUNDARY</td>
<td>4</td>
<td>Use clipping boundary</td>
</tr>
<tr>
<td>Image.USE_TRANSPARENCY</td>
<td>8</td>
<td>Transparency is on</td>
</tr>
</tbody>
</table>

dxf.clipping
Clipping state:

<table>
<thead>
<tr>
<th>Value</th>
<th>Clipping State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>clipping off</td>
</tr>
<tr>
<td>1</td>
<td>clipping on</td>
</tr>
</tbody>
</table>

dxfbrightness
Brightness value (0-100; default = 50)
ezdxf Documentation, Release 0.16.2

dxf . contrast
Contrast value (0-100; default = 50)

dxf . fade
Fade value (0-100; default = 0)

dxf . clipping_boundary_type
Clipping boundary type:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Rectangular</td>
</tr>
<tr>
<td>2</td>
<td>Polygonal</td>
</tr>
</tbody>
</table>


dxf . count_boundary_points
Number of clip boundary vertices, maintained by ezdxf.

dxf . clip_mode
Clip mode (DXF R2010):

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Outside</td>
</tr>
<tr>
<td>1</td>
<td>Inside</td>
</tr>
</tbody>
</table>

boundary_path
A list of vertices as pixel coordinates, Two vertices describe a rectangle, lower left corner is (-0.5, -0.5) and upper right corner is (ImageSizeX-0.5, ImageSizeY-0.5), more than two vertices is a polygon as clipping path. All vertices as pixel coordinates. (read/write)

image_def
Returns the associated IMAGEDEF entity, see ImageDef.

reset_boundary_path () → None
Reset boundary path to the default rectangle [(-0.5, -0.5), (ImageSizeX-0.5, ImageSizeY-0.5)].

set_boundary_path (vertices: Iterable[Vertex]) → None
Set boundary path to vertices. Two vertices describe a rectangle (lower left and upper right corner), more than two vertices is a polygon as clipping path.

boundary_path_wcs () → List[Vec3]
Returns the boundary/clipping path in WCS coordinates.

New in version 0.14.

Since version 0.16 it’s recommended to create the clipping path as Path object by the make_path() function:

```python
from ezdxf.path import make_path
image = ...  # get image entity
clipping_path = make_path(image)
```

transform (m: Matrix44) → Image
Transform IMAGE entity by transformation matrix m inplace.

Leader

The LEADER entity (DXF Reference) represents an arrow, made up of one or more vertices (or spline fit points) and an arrowhead. The label or other content to which the Leader is attached is stored as a separate entity, and is not part of the Leader itself.

6.8. Reference 243
Leader shares its styling infrastructure with Dimension.

By default a Leader without any annotation is created. For creating more fancy leaders and annotations see documentation provided by Autodesk or Demystifying DXF: LEADER and MULTILEADER implementation notes.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'LEADER'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_leader()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

class ezdxf.entities.Leader

dxf.dimstyle
   Name of Dimstyle as string.

dxf.has_arrowhead
   
   0 Disabled
   1 Enabled


dxf.path_type
   Leader path type:

   0 Straight line segments
   1 Spline


dxf.annotation_type

   0 Created with text annotation
   1 Created with tolerance annotation
   2 Created with block reference annotation
   3 Created without any annotation (default)


dxf.hookline_direction
   Hook line direction flag:

   0 Hookline (or end of tangent for a splined leader) is the opposite direction from the horizontal vector
   1 Hookline (or end of tangent for a splined leader) is the same direction as horizontal vector (see has_hook_line)


dxf.has_hookline

   0 No hookline
   1 Has a hookline


dxf.text_height
   Text annotation height in drawing units.
dxf.text_width
Text annotation width.

dxf.block_color
Color to use if leader’s DIMCLRDRD = BYBLOCK

dxf.annotation_handle
Hard reference (handle) to associated annotation (MText, Tolerance, or Insert entity)

dxf.normal_vector
Extrusion vector? default = (0, 0, 1).

dxf.horizontal_direction
Horizontal direction for leader, default = (1, 0, 0).

dxf.leader_offset_block_ref
Offset of last leader vertex from block reference insertion point, default = (0, 0, 0).

dxf.leader_offset_annotation_placement
Offset of last leader vertex from annotation placement point, default = (0, 0, 0).

vertices
List of Vec3 objects, representing the vertices of the leader (3D Point in WCS).

set_vertices(vertices: Iterable[Vertex])
Set vertices of the leader, vertices is an iterable of (x, y [,z]) tuples or Vec3.

transform(m: Matrix44) → Leader
Transform LEADER entity by transformation matrix m inplace.

virtual_entities() → Iterable[Union[Line, Arc]]
Yields ‘virtual’ parts of LEADER as DXF primitives.

This entities are located at the original positions, but are not stored in the entity database, have no handle and are not assigned to any layout.

New in version 0.14.

explode(target_layout: BaseLayout = None) → EntityQuery
Explode parts of LEADER as DXF primitives into target layout, if target layout is None, the target layout is the layout of the LEADER.

Returns an EntityQuery container with all DXF parts.

Parameters target_layout – target layout for DXF parts, None for same layout as source entity.

New in version 0.14.

Line

LINE (DXF Reference) entity is a 3D line from Line.dxf.start to Line.dxf.end.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'LINE'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_line()</td>
</tr>
<tr>
<td>Inherited DXF Attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
</tbody>
</table>
Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.Line

    dxf.start
    start point of line (2D/3D Point in WCS)
    
    dxf.end
    end point of line (2D/3D Point in WCS)
    
    dxf.thickness
    Line thickness in 3D space in direction extrusion, default value is 0. This value should not be confused with the lineweight value.
    
    dxf.extrusion
    extrusion vector, default value is (0, 0, 1)
    
    transform(m: Matrix44) → Line
    Transform the LINE entity by transformation matrix m inplace.
    
    translate(dx: float, dy: float, dz: float) → Line
    Optimized LINE translation about dx in x-axis, dy in y-axis and dz in z-axis.

LWPolyline

The LWPOLYLINE entity (DXF Reference) is defined as a single graphic entity, which differs from the old-style Polyline entity, which is defined as a group of sub-entities. LWPolyline display faster (in AutoCAD) and consume less disk space, it is a planar element, therefore all points are located in the OCS as (x, y)-tuples (LWPolyline.dxf.elevation is the z-axis value).

Changed in version 0.8.9: LWPolyline stores point data as packed data (array.array).

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'LWPOLYLINE'</td>
</tr>
<tr>
<td>factory function</td>
<td>add_lwpolyline()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

Bulge value

The bulge value is used to create arc shaped line segments for Polyline and LWPolyline entities. The arc starts at the vertex which includes the bulge value and ends at the following vertex. The bulge value defines the ratio of the arc sagitta (versine) to half line segment length, a bulge value of 1 defines a semicircle.

The sign of the bulge value defines the side of the bulge:

- positive value (> 0): bulge is right of line (counter clockwise)
- negative value (< 0): bulge is left of line (clockwise)
- 0 = no bulge
Start- and end width

The start width and end width values defines the width in drawing units for the following line segment. To use the default width value for a line segment set value to 0.

Width and bulge values at last point

The width and bulge values of the last point has only a meaning if the polyline is closed, and they apply to the last line segment from the last to the first point.

See also:

Tutorial for LWPolyline and Bulge Related Functions
User Defined Point Format Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Point Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x-coordinate</td>
</tr>
<tr>
<td>y</td>
<td>y-coordinate</td>
</tr>
<tr>
<td>s</td>
<td>start width</td>
</tr>
<tr>
<td>e</td>
<td>end width</td>
</tr>
<tr>
<td>b</td>
<td>bulge value</td>
</tr>
<tr>
<td>v</td>
<td>(x, y [, z]) as tuple</td>
</tr>
</tbody>
</table>

**class** `ezdxf.entities.LWPolyline`

```python
dxf.elevation

OCS z-axis value for all polyline points, default=0
```

```python
dxf.flags

Constants defined in `ezdxf.lldxf.const`:
```
```

<table>
<thead>
<tr>
<th>dxf.flags</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LWPOLYLINE_CLOSED</td>
<td>1</td>
<td>polyline is closed</td>
</tr>
<tr>
<td>LWPOLYLINE_PLINEGEN</td>
<td>128</td>
<td>???</td>
</tr>
</tbody>
</table>
```

```python
dxf.const_width

Constant line width (float), default value is 0.
```

```python
dxf.count

Count of polyline points (read only), same as `len(polyline)`
```

```python
close (state: bool = True) → None

Get/set closed state of LWPOLYLINE. Compatibility interface to `Polyline`
```

```python
__len__() → int

Returns count of polyline points.
```

```python
__getitem__(index: int) → Tuple[float, float, float, float, float]

Returns point at position `index` as (x, y, start_width, end_width, bulge) tuple. start_width, end_width and bulge is 0 if not present, supports extended slicing. Point format is fixed as 'xyseb'. All coordinates in OCS.
```

```python
__setitem__(index: int, value: Sequence[float]) → None

Set point at position `index` as (x, y, [start_width, [end_width, [bulge]]) tuple. If start_width or end_width is 0 or left off the default value is used. If the bulge value is left off, bulge is 0 by default (straight line). Does NOT support extend slicing. Point format is fixed as 'xyseb'. All coordinates in OCS.
```

**Parameters**

- **index** – point index
- **value** – point value as (x, y, [start_width, [end_width, [bulge]]) tuple

```python
__delitem__(index: int) → None

Delete point at position `index`, supports extended slicing.
```

```python
__iter__() → Iterable[Tuple[float, float, float, float, float]]

Returns iterable of tuples (x, y, start_width, end_width, bulge).
```
vertices () → Iterable[ Tuple[ float, float ]]
Returns iterable of all polyline points as (x, y) tuples in OCS (dxf.elevation is the z-axis value).

vertices_in_wcs () → Iterable[ Vertex]
Returns iterable of all polyline points as Vec3(x, y, z) in WCS.

append( point: Sequence[ float ], format: str = 'xyseb' ) → None
Append point to polyline, format specifies a user defined point format.
All coordinates in OCS.

Parameters

- point – (x, y, [start_width, [end_width, [bulge]]]) tuple
- format – format string, default is 'xyseb', see: format codes

append_points( points: Iterable[ Sequence[ float ]], format: str = 'xyseb' ) → None
Append new points to polyline, format specifies a user defined point format.
All coordinates in OCS.

Parameters

- points – iterable of point, point is (x, y, [start_width, [end_width, [bulge]]]) tuple
- format – format string, default is 'xyseb', see: format codes

insert( pos: int, point: Sequence[ float ], format: str = 'xyseb' ) → None
Insert new point in front of positions pos, format specifies a user defined point format.
All coordinates in OCS.

Parameters

- pos – insert position
- point – point data
- format – format string, default is 'xyseb', see: format codes

clear() → None
Remove all points.

get_points( format: str = 'xyseb' ) → List[ Sequence[ float ]]
Returns all points as list of tuples, format specifies a user defined point format.
All points in OCS as (x, y) tuples (dxf.elevation is the z-axis value).

Parameters

- format – format string, default is 'xyseb', see format codes

set_points( points: Iterable[ Sequence[ float ]], format: str = 'xyseb' ) → None
Remove all points and append new points.
All coordinates in OCS.

Parameters

- points – iterable of point, point is (x, y, [start_width, [end_width, [bulge]]]) tuple
- format – format string, default is 'xyseb', see format codes

points( format: str = 'xyseb' ) → List[ Sequence[ float ]]
Context manager for polyline points. Returns a standard Python list of points, according to the format string.
All coordinates in OCS.
Parameters **format** – format string, see *format codes*

**transform** \((m: \text{Matrix44}) \rightarrow \text{LWPolyline}\)
Transform the LWPOLYLINE entity by transformation matrix \(m\) inplace.

**virtual_entities** () \(\rightarrow\) Iterable[Union[Line, Arc]]
Yields ‘virtual’ parts of LWPOLYLINE as LINE or ARC entities.
This entities are located at the original positions, but are not stored in the entity database, have no handle and are not assigned to any layout.

**explode** (target_layout: BaseLayout = None) \(\rightarrow\) EntityQuery
Explode parts of LWPOLYLINE as LINE or ARC entities into target layout, if target layout is None, the target layout is the layout of the LWPOLYLINE.

Returns an EntityQuery container with all DXF parts.

**Parameters target_layout** – target layout for DXF parts, None for same layout as source entity.

**MLine**

The MLINE entity (DXF Reference).

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'MLINE'</td>
</tr>
<tr>
<td>factory function</td>
<td>add_mline()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

**class** ezdxf.entities.MLine

**dxf.style_name**
MLLineStyle name stored in Drawing.mline_styles dictionary, use set_style() to change the MLINESTYLE and update geometry accordingly.

**dxf.style_handle**
Handle of MLLineStyle, use set_style() to change the MLINESTYLE and update geometry accordingly.

**dxf.scale_factor**
MLINE scaling factor, use method set_scale_factor() to change the scaling factor and update geometry accordingly.

**dxf.justification**
Justification defines the location of the MLINE in relation to the reference line, use method set_justification() to change the justification and update geometry accordingly.

Constants defined in ezdxf.lldxf.const:

<table>
<thead>
<tr>
<th>dxf.justification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLINE_TOP</td>
<td>0</td>
</tr>
<tr>
<td>MLINE_ZERO</td>
<td>1</td>
</tr>
<tr>
<td>MLINE_BOTTOM</td>
<td>2</td>
</tr>
<tr>
<td>MLINE_RIGHT</td>
<td>0</td>
</tr>
<tr>
<td>MLINE_CENTER</td>
<td>1</td>
</tr>
<tr>
<td>MLINE_LEFT</td>
<td>2</td>
</tr>
</tbody>
</table>
**dxflflags**

Use method `close()` and the properties `start_caps` and `end_caps` to change these flags.

Constants defined in `ezdxf.lldxf.const`:

<table>
<thead>
<tr>
<th>dxflflags</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLINE_HAS_VERTEX</td>
<td>1</td>
</tr>
<tr>
<td>MLINE_CLOSED</td>
<td>2</td>
</tr>
<tr>
<td>MLINE_SUPPRESS_START_CAPS</td>
<td>4</td>
</tr>
<tr>
<td>MLINE_SUPPRESS_END_CAPS</td>
<td>8</td>
</tr>
</tbody>
</table>

**dxf.start_location**

Start location of the reference line. (read only)

**dxf.count**

Count of MLINE vertices. (read only)

**dxf.style_element_count**

Count of elements in `MLineStyle` definition. (read only)

**dxf.extrusion**

Normal vector of the entity plane, but MLINE is not an OCS entity, all vertices of the reference line are WCS! (read only)

**vertices**

MLINE vertices as `MLineVertex` objects, stored in a regular Python list.

**set_style**(name: `str`) → None

Set MLINESTYLE by name and update geometry accordingly. The MLINESTYLE definition must exist.

**set_scale_factor**(value: `float`) → None

Set the scale factor and update geometry accordingly.

**set_justification**(value: `int`) → None

Set MLINE justification and update geometry accordingly. See `dxf.justification` for valid settings.

**close**(state: `bool` = `True`) → None

Get/set closed state of MLINE and update geometry accordingly. Compatibility interface to `Polyline`

**__len__()**

Count of MLINE vertices.

**start_location()** → `Vec3`

Returns the start location of the reference line. Callback function for `dxf.start_location`.

**get_locations()** → `List[Vec3]`

Returns the vertices of the reference line.

**extend**(vertices: `Iterable[Vec3]`) → None

Append multiple vertices to the reference line.

It is possible to work with 3D vertices, but all vertices have to be in the same plane and the normal vector of this plan is stored as extrusion vector in the MLINE entity.

**clear()** → None

Remove all MLINE vertices.

**update_geometry()** → None

Regenerate the MLINE geometry based on current settings.

**generate_geometry**(vertices: `List[ezdxf.math._vector.Vec3]`) → None

Regenerate the MLINE geometry for new reference line defined by `vertices`. 
**transform**\((m: \text{Matrix44}) \rightarrow \text{MLine}\)

Transform MLINE entity by transformation matrix \(m\) inplace.

**virtual_entities**\() \rightarrow \text{Iterable}[\text{DXFGraphic}]\)

Yields ‘virtual’ parts of MLINE as LINE, ARC and HATCH entities.

This entities are located at the original positions, but are not stored in the entity database, have no handle and are not assigned to any layout.

**explode**\((\text{target_layout: BaseLayout} = \text{None}) \rightarrow \text{EntityQuery}\)

Explode parts of MLINE as LINE, ARC and HATCH entities into target layout, if target layout is \text{None}, the target layout is the layout of the MLINE.

Returns an \text{EntityQuery} container with all DXF parts.

**Parameters**

- **target_layout**
  - target layout for DXF parts, \text{None} for same layout as source entity.

**class ezdxf.entities.MLineVertex**

- **location**
  Reference line vertex location.

- **line_direction**
  Reference line direction.

- **miter_direction**

- **line_params**
  The line parameterization is a list of float values. The list may contain zero or more items.

  The first value (miter-offset) is the distance from the vertex \text{location} along the \text{miter_direction} vector to the point where the line element’s path intersects the miter vector.

  The next value (line-start-offset) is the distance along the \text{line_direction} from the miter/line path intersection point to the actual start of the line element.

  The next value (dash-length) is the distance from the start of the line element (dash) to the first break (gap) in the line element. The successive values continue to list the start and stop points of the line element in this segment of the mline.

- **fill_params**
  The fill parameterization is also a list of float values. Similar to the line parameterization, it describes the parameterization of the fill area for this mline segment. The values are interpreted identically to the line parameters and when taken as a whole for all line elements in the mline segment, they define the boundary of the fill area for the mline segment.

**class ezdxf.entities.MLineStyle**

The \text{MLineStyle} stores the style properties for the MLINE entity.

- **dxf.name**
- **dxf.description**
- **dxf.flags**
- **dxf.fill_color**
  \text{AutoCAD Color Index (ACI) value of the fill color}
- **dxf.start_angle**
- **dxf.end_angle**
**elements**

*MLLineStyleElements* object

**update_all()**

Update all MLINE entities using this MLINESTYLE.

The update is required if elements were added or removed or the offset of any element was changed.

class ezdxf.entities.mline.MLineStyleElements

**elements**

List of *MLLineStyleElement* objects, one for each line element.

MLLineStyleElements.__len__()

MLLineStyleElements.__getitem__(item)

MLLineStyleElements.append(offset: float, color: int = 0, linetype: str = 'BYLAYER') → None

Append a new line element.

**Parameters**

- **offset** – normal offset from the reference line: if justification is MLINE_ZERO, positive values are above and negative values are below the reference line.
- **color** – *AutoCAD Color Index (ACI)* value
- **linetype** – linetype name

class ezdxf.entities.mline.MLineStyleElement

Named tuple to store properties of a line element.

**offset**

Normal offset from the reference line: if justification is MLINE_ZERO, positive values are above and negative values are below the reference line.

**color**

*AutoCAD Color Index (ACI)* value

**linetype**

Linetype name

**Mesh**

The MESH entity (DXF Reference) is a 3D mesh similar to the *Polyface* entity.

All vertices in *WCS* as (x, y, z) tuples

Changed in version 0.8.9: *Mesh* stores vertices, edges, faces and creases as packed data.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'MESH'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_mesh()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td><em>Common graphical DXF attributes</em></td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

See also:

*Tutorial for Mesh* and helper classes: *MeshBuilder, MeshVertexMerger*
**Mesh**

```python
class ezdxf.entities.Mesh
```

dxf.version

dxf.blend_crease

0 = off, 1 = on

dxf.subdivision_levels

0 for no smoothing else integer greater than 0.

vertices

Vertices as list like `VertexArray`. (read/write)

edges

Edges as list like `TagArray`. (read/write)

faces

Faces as list like `TagList`. (read/write)

creases

Creases as `array.array`. (read/write)

edit_data() → ezdxf.entities.mesh.MeshData

Context manager various mesh data, returns `MeshData`.

Despite that vertices, edge and faces since `ezdxf` v0.8.9 are accessible as packed data types, the usage of `MeshData` by context manager `edit_data()` is still recommended.

transform(m: Matrix44) → Mesh

Transform the MESH entity by transformation matrix m inplace.

**MeshData**

```python
class ezdxf.entities.MeshData
```

vertices

A standard Python list with (x, y, z) tuples (read/write)

faces

A standard Python list with (v1, v2, v3,...) tuples (read/write)

Each face consist of a list of vertex indices (= index in vertices).

edges

A standard Python list with (v1, v2) tuples (read/write)

Each edge consist of exact two vertex indices (= index in vertices).

edge_crease_values

A standard Python list of float values, one value for each edge. (read/write)

add_face(vertices: Iterable[Sequence[float]]) → Sequence[int]

Add a face by coordinates, vertices is a list of (x, y, z) tuples.

add_edge(vertices: Sequence[Sequence[float]]) → Sequence[int]

Add an edge by coordinates, vertices is a list of two (x, y, z) tuples.

optimize(precision: int = 6)

Try to reduce vertex count by merging near vertices. precision defines the decimal places for coordinate be equal to merge two vertices.
**MText**

The MTEXT entity (DXF Reference) fits a multiline text in a specified width but can extend vertically to an indefinite length. You can format individual words or characters within the MText.

See also:

* Tutorial for MText

<table>
<thead>
<tr>
<th>Subclass of</th>
<th><code>ezdxf.entities.DXFGraphic</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'MTEXT'</td>
</tr>
<tr>
<td>Factory function</td>
<td><code>ezdxf.layouts.BaseLayout.add_mtext()</code></td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td><code>Common graphical DXF attributes</code></td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

```python
class ezdxf.entities.MText

dxf.insert
   Insertion point (3D Point in OCS)

dxf.char_height
   Initial text height (float); default=1.0

dxf.width
   Reference text width (float), forces text wrapping at given width.

dxf.attachment_point
   Constants defined in `ezdxf.lldxf.const`:

<table>
<thead>
<tr>
<th>MText.dxf.attachment_point</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTEXT_TOP_LEFT</td>
<td>1</td>
</tr>
<tr>
<td>MTEXT_TOP_CENTER</td>
<td>2</td>
</tr>
<tr>
<td>MTEXT_TOP_RIGHT</td>
<td>3</td>
</tr>
<tr>
<td>MTEXT_MIDDLE_LEFT</td>
<td>4</td>
</tr>
<tr>
<td>MTEXT_MIDDLE_CENTER</td>
<td>5</td>
</tr>
<tr>
<td>MTEXT_MIDDLE_RIGHT</td>
<td>6</td>
</tr>
<tr>
<td>MTEXT_BOTTOM_LEFT</td>
<td>7</td>
</tr>
<tr>
<td>MTEXT_BOTTOM_CENTER</td>
<td>8</td>
</tr>
<tr>
<td>MTEXT_BOTTOM_RIGHT</td>
<td>9</td>
</tr>
</tbody>
</table>
```

```python
dxf.flow_direction
   Constants defined in `ezdxf.const`:

<table>
<thead>
<tr>
<th>MText.dxf.flow_direction</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTEXT_LEFT_TO_RIGHT</td>
<td>1</td>
<td>left to right</td>
</tr>
<tr>
<td>MTEXT_TOP_TO_BOTTOM</td>
<td>43</td>
<td>top to bottom</td>
</tr>
<tr>
<td>MTEXT_BY_STYLE</td>
<td>5</td>
<td>by style (the flow direction is inherited from the associated text style)</td>
</tr>
</tbody>
</table>
```

dxf.style
   Text style (string); default = 'STANDARD'

---

6.8. Reference 255
ezdxf Documentation, Release 0.16.2

dxftext_direction
X-axis direction vector in WCS (3D Point); default value is (1, 0, 0); if dxf.rotation and dxf.text_direction are present, dxf.text_direction wins.

dxf.rotation
Text rotation in degrees (float); default = 0

dxfline_spacing_style
Line spacing style (int), see table below

dxf.line_spacing_factor
Percentage of default (3-on-5) line spacing to be applied. Valid values range from 0.25 to 4.00 (float).

Constants defined in ezdxf.lldxf.const:

<table>
<thead>
<tr>
<th>MText.dxf.line_spacing_style</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTEXT_AT_LEAST</td>
<td>1</td>
<td>taller characters will override</td>
</tr>
<tr>
<td>MTEXT_EXACT</td>
<td>2</td>
<td>taller characters will not override</td>
</tr>
</tbody>
</table>

dxfgbg_fill
Defines the background fill type. (DXF R2007)

<table>
<thead>
<tr>
<th>MText.dxf.bg_fill</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTEXT_BG_OFF</td>
<td>0</td>
<td>no background color</td>
</tr>
<tr>
<td>MTEXT_BG_COLOR</td>
<td>1</td>
<td>use specified color</td>
</tr>
<tr>
<td>MTEXT_BG_WINDOW_COLOR</td>
<td>2</td>
<td>use window color (?)</td>
</tr>
<tr>
<td>MTEXT_BG_CANVAS_COLOR</td>
<td>3</td>
<td>use canvas background color</td>
</tr>
</tbody>
</table>

dxflexbox_fill_scale
Determines how much border there is around the text. (DXF R2007)

Requires: bg_fill, bg_fill_color else AutoCAD complains

Better use set_bg_color()

dxf.bg_fill_color
Background fill color as AutoCAD Color Index (ACI) (DXF R2007)

Better use set_bg_color()

dxf.bg_fill_true_color
Background fill color as true color value (DXF R2007), also dxf.bg_fill_color must be present, else AutoCAD complains.

Better use set_bg_color()

dxf.bg_fill_color_name
Background fill color as name string (?) (DXF R2007), also dxf.bg_fill_color must be present, else AutoCAD complains.

Better use set_bg_color()

dxf.transparency
Transparency of background fill color (DXF R2007), not supported by AutoCAD or BricsCAD.

text
MTEXT content as string (read/write).

Line endings \n will be replaced by the MTEXT line endings \P at DXF export, but not vice versa \P by \n at DXF file loading.
**set_location** (<i>insert: Vertex, rotation: float = None, attachment_point: int = None</i>) → MText

Set attributes `dxf.insert`, `dxf.rotation` and `dxf.attachment_point`, None for `dxf.rotation` or `dxf.attachment_point` preserves the existing value.

**get_rotation** () → float

Get text rotation in degrees, independent if it is defined by `dxf.rotation` or `dxf.text_direction`.

**set_rotation** (<i>angle: float</i>) → ezdxf.entities.mtext.MText

Set attribute rotation to `angle` (in degrees) and deletes `dxf.text_direction` if present.

**set_bg_color** (<i>color: Union[int, str, Tuple[int, int, int], None], scale: float = 1.5</i>)

Set background color as *AutoCAD Color Index (ACI)* value or as name string or as RGB tuple (r, g, b).

Use special color name `canvas`, to set background color to canvas background color.

**Parameters**

- **color** – color as *AutoCAD Color Index (ACI)*, string or RGB tuple
- **scale** – determines how much border there is around the text, the value is based on the text height, and should be in the range of [1, 5], where 1 fits exact the MText entity.

---

**iadd__(text: str)** → MText

Append text to existing content (`text` attribute).

**append**(text: str) → MText

Append text to existing content (`text` attribute).

**set_font** (<i>name: str, bold: bool = False, italic: bool = False, codepage: int = 1252, pitch: int = 0</i>) → None

Append font change (e.g. ‘\Fkroeger|b0|i0|c238|p10’ ) to existing content (`text` attribute).

**Parameters**

- **name** – font name
- **bold** – flag
- **italic** – flag
- **codepage** – character codepage
- **pitch** – font size

**set_color** (<i>color_name: str</i>) → None

Append text color change to existing content, `color_name` as red, yellow, green, cyan, blue, magenta or white.

**add_stacked_text** (<i>upr: str, lwr: str, t: str = '^'</i>) → None

Add stacked text `upr` over `lwr`, `t` defines the kind of stacking:

```
****: vertical stacked without divider line, e.g. \SA^B:
  A
  B

/\**: vertical stacked with divider line, e.g. \SX/Y:
  X
  -
  Y

###**: diagonal stacked, with slanting divider line, e.g. \SI#4:
  1/4
```
plain_text (split=False) → Union[List[str], str]
Returns text content without formatting codes.

Parameters split – returns list of strings splitted at line breaks if True else returns a single string.

transform (m: Matrix44) → MText
Transform the MTEXT entity by transformation matrix m inplace.

MText Inline Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\L</td>
<td>Start underline</td>
</tr>
<tr>
<td>\l</td>
<td>Stop underline</td>
</tr>
<tr>
<td>\O</td>
<td>Start overstrike</td>
</tr>
<tr>
<td>\o</td>
<td>Stop overstrike</td>
</tr>
<tr>
<td>\K</td>
<td>Start strike-through</td>
</tr>
<tr>
<td>\k</td>
<td>Stop strike-through</td>
</tr>
<tr>
<td>\P</td>
<td>New paragraph (new line)</td>
</tr>
<tr>
<td>\pxi</td>
<td>Control codes for bullets, numbered paragraphs and columns</td>
</tr>
<tr>
<td>\X</td>
<td>Paragraph wrap on the dimension line (only in dimensions)</td>
</tr>
<tr>
<td>\Q</td>
<td>Slanting (obliquing) text by angle - e.g. \Q30;</td>
</tr>
<tr>
<td>\H</td>
<td>Text height - e.g. \H3x;</td>
</tr>
<tr>
<td>\W</td>
<td>Text width - e.g. \W0.8x;</td>
</tr>
<tr>
<td>\F</td>
<td>Font selection e.g. \Fgdt;o - GDT-tolerance</td>
</tr>
<tr>
<td>\S</td>
<td>Stacking, fractions e.g. \SA^B or \SXY or \S1#4</td>
</tr>
<tr>
<td>\A</td>
<td>Alignment</td>
</tr>
<tr>
<td>\C</td>
<td>Color change</td>
</tr>
<tr>
<td>\T</td>
<td>Tracking, char.spacing - e.g. \T2;</td>
</tr>
<tr>
<td>~</td>
<td>Non-wrapping space, hard space</td>
</tr>
<tr>
<td>{}</td>
<td>Braces - define the text area influenced by the code, codes and braces can be nested up to 8 levels deep</td>
</tr>
<tr>
<td>\</td>
<td>Escape character - e.g. \ = “\”</td>
</tr>
</tbody>
</table>
Convenient constants defined in MText:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNDERLINE_START</td>
<td>start underline text (b += b.UNDERLINE_START)</td>
</tr>
<tr>
<td>UNDERLINE_STOP</td>
<td>stop underline text (b += b.UNDERLINE_STOP)</td>
</tr>
<tr>
<td>UNDERLINE</td>
<td>underline text (b += b.UNDERLINE % &quot;Text&quot;)</td>
</tr>
<tr>
<td>OVERSTRIKE_START</td>
<td>start overstrike</td>
</tr>
<tr>
<td>OVERSTRIKE_STOP</td>
<td>stop overstrike</td>
</tr>
<tr>
<td>OVERSTRIKE</td>
<td>overstrike text</td>
</tr>
<tr>
<td>STRIKE_START</td>
<td>start strike trough</td>
</tr>
<tr>
<td>STRIKE_STOP</td>
<td>stop strike trough</td>
</tr>
<tr>
<td>STRIKE</td>
<td>strike trough text</td>
</tr>
<tr>
<td>GROUP_START</td>
<td>start of group</td>
</tr>
<tr>
<td>GROUP_END</td>
<td>end of group</td>
</tr>
<tr>
<td>GROUP</td>
<td>group text</td>
</tr>
<tr>
<td>NEW_LINE</td>
<td>start in new line (b += &quot;Text.&quot; + b.NEW_LINE)</td>
</tr>
<tr>
<td>NBSP</td>
<td>none breaking space (b += &quot;Python&quot; + b.NBSP + &quot;3.4&quot;)</td>
</tr>
</tbody>
</table>

Point

POINT (DXF Reference) at location `dxf.location`.

The POINT styling is a global setting, stored as header variable `SPDMODE`, this also means all POINT entities in a DXF document have the same styling:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>center dot (.)</td>
</tr>
<tr>
<td>1</td>
<td>none ( )</td>
</tr>
<tr>
<td>2</td>
<td>cross (+)</td>
</tr>
<tr>
<td>3</td>
<td>x-cross (x)</td>
</tr>
<tr>
<td>4</td>
<td>tick (')</td>
</tr>
</tbody>
</table>

Combined with these bit values

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>circle</td>
</tr>
<tr>
<td>64</td>
<td>Square</td>
</tr>
</tbody>
</table>

e.g. circle + square + center dot = 32 + 64 + 0 = 96

The size of the points is defined by the header variable `PDSIZE`:
0 | 5% of draw area height  
<0 | Specifies a percentage of the viewport size  
>0 | Specifies an absolute size

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'POINT'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_point()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
</tbody>
</table>

**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.Point

dxf.location  
Location of the point (2D/3D Point in WCS)

dxf.angle  
Angle in degrees of the x-axis for the UCS in effect when POINT was drawn (float); used when PDMODE is nonzero.

transform (m: Matrix44) → Point  
Transform the POINT entity by transformation matrix m inplace.

translate (dx: float, dy: float, dz: float) → Point  
Optimized POINT translation about dx in x-axis, dy in y-axis and dz in z-axis.

virtual_entities (pdsize: float = 1, pdmode: int = 0) → List[DXFGraphic]  
Yields point graphic as DXF primitives LINE and CIRCLE entities. The dimensionless point is rendered as zero-length line!

Check for this condition:
```
e.dxftype() == 'LINE' and e.dxf.start.isclose(e.dxf.end)
```

if the rendering engine can’t handle zero-length lines.

**Parameters**

- **pdsize** – point size in drawing units
- **pdmode** – point styling mode

New in version 0.15.

**Polyline**

The POLYLINE entity (POLYLINE DXF Reference) is very complex, it’s used to build 2D/3D polylines, 3D meshes and 3D polyfaces. For every type exists a different wrapper class but they all have the same dxftype of 'POLYLINE'. Detect POLYLINE type by Polyline.get_mode().

POLYLINE types returned by Polyline.get_mode():

- 'AcDb2dPolyline' for 2D Polyline
- 'AcDb3dPolyline' for 3D Polyline
• 'AcDbPolygonMesh' for `Polymesh`
• 'AcDbPolyFaceMesh' for `Polyface`

For 2D entities all vertices in OCS.
For 3D entities all vertices in WCS.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th><code>ezdxf.entities.DXFGraphic</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'POLYLINE'</td>
</tr>
<tr>
<td>2D factory function</td>
<td><code>ezdxf.layouts.BaseLayout.add_polyline2d()</code></td>
</tr>
<tr>
<td>3D factory function</td>
<td><code>ezdxf.layouts.BaseLayout.add_polyline3d()</code></td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td><code>Common graphical DXF attributes</code></td>
</tr>
</tbody>
</table>

**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!

class `ezdxf.entities.Polyline`

Vertex entities are stored in a standard Python list `Polyline.vertices`. Vertices can be retrieved and deleted by direct access to `Polyline.vertices` attribute:

```python
# delete first and second vertex
del polyline.vertices[:2]
```

dxf.**elevation**

Elevation point, the X and Y values are always 0, and the Z value is the polyline’s elevation (3D Point in OCS when 2D, WCS when 3D).

dxf.**flags**

Constants defined in `ezdxf.lldxf.const`:

<table>
<thead>
<tr>
<th>Polyline.dxf.flags</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLYLINE_CLOSED</td>
<td>1</td>
<td>This is a closed Polyline (or a polygon mesh closed in the M direction)</td>
</tr>
<tr>
<td>POLYLINE_MESH_CLOSED_M_DIRECTION</td>
<td>1</td>
<td>equals POLYLINE_CLOSED</td>
</tr>
<tr>
<td>POLYLINE_CURVEFIT_VERTICES_ADDED</td>
<td>2</td>
<td>Curve-fit vertices have been added</td>
</tr>
<tr>
<td>POLYLINE_SPLINEFIT_VERTICES_ADDED</td>
<td>4</td>
<td>Spline-fit vertices have been added</td>
</tr>
<tr>
<td>POLYLINE_3D_POLYLINE</td>
<td>8</td>
<td>This is a 3D Polyline</td>
</tr>
<tr>
<td>POLYLINE_3D_POLYMESH</td>
<td>16</td>
<td>This is a 3D polygon mesh</td>
</tr>
<tr>
<td>POLYLINE_MESH_CLOSED_N_DIRECTION</td>
<td>32</td>
<td>The polygon mesh is closed in the N direction</td>
</tr>
<tr>
<td>POLYLINE_POLYFACE_MESH</td>
<td>64</td>
<td>This Polyline is a polyface mesh</td>
</tr>
<tr>
<td>POLYLINE_GENERATE_LINETYPE_PATTERN</td>
<td>128</td>
<td>The linetype pattern is generated continuously around the vertices of this Polyline</td>
</tr>
</tbody>
</table>

dxf.**default_start_width**

Default line start width (float); default = 0

dxf.**default_end_width**

Default line end width (float); default = 0
dxm.m_count
  Polymesh M vertex count (int); default = 1

dxm.n_count
  Polymesh N vertex count (int); default = 1

dxm.m_smooth_density
  Smooth surface M density (int); default = 0

dxm.n_smooth_density
  Smooth surface N density (int); default = 0

dxm.smooth_type
  Curves and smooth surface type (int); default=0, see table below

  Constants for smooth_type defined in ezdxf.lldxf.const:

<table>
<thead>
<tr>
<th>Polyline.dxf.smooth_type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLYMESH_NO_SMOOTH</td>
<td>0</td>
<td>no smooth surface fitted</td>
</tr>
<tr>
<td>POLYMESH_QUADRATIC_BSPLINE</td>
<td>5</td>
<td>quadratic B-spline surface</td>
</tr>
<tr>
<td>POLYMESH_CUBIC_BSPLINE</td>
<td>6</td>
<td>cubic B-spline surface</td>
</tr>
<tr>
<td>POLYMESH_BEZIER_SURFACE</td>
<td>8</td>
<td>Bezier surface</td>
</tr>
</tbody>
</table>

vertices
  List of Vertex entities.

is_2d_polyline
  True if POLYLINE is a 2D polyline.

is_3d_polyline
  True if POLYLINE is a 3D polyline.

is_polygon_mesh
  True if POLYLINE is a polygon mesh, see Polymesh

is_poly_face_mesh
  True if POLYLINE is a poly face mesh, see Polyface

is_closed
  True if POLYLINE is closed.

is_m_closed
  True if POLYLINE (as Polymesh) is closed in m direction.

is_n_closed
  True if POLYLINE (as Polymesh) is closed in n direction.

has_arc
  Returns True if 2D POLYLINE has an arc segment.

has_width
  Returns True if 2D POLYLINE has default width values or any segment with width attributes.

  New in version 0.14.

get_mode() → str
  Returns POLYLINE type as string:
  - ‘AcDb2dPolyline’
  - ‘AcDb3dPolyline’
  - ‘AcDbPolygonMesh’
• ‘AcDbPolyFaceMesh’

m_close(status=True) → None
Close POLYMESH in m direction if status is True (also closes POLYLINE), clears closed state if status is False.

n_close(status=True) → None
Close POLYMESH in n direction if status is True, clears closed state if status is False.

close(m_close=True, n_close=False) → None
Set closed state of POLYMESH and POLYLINE in m direction and n direction. True set closed flag, False clears closed flag.

__len__() → int
Returns count of Vertex entities.

__getitem__(pos) → ezdxf.entities.polyline.DXFVertex
Get Vertex entity at position pos, supports list slicing.

points() → Iterable[ezdxf.math._vector.Vec3]
Returns iterable of all polyline vertices as (x, y, z) tuples, not as Vertex objects.

append_vertex(point: Vertex, dxfattribs: dict = None) → None
Append a single Vertex entity at location point.

Parameters

• point – as (x, y[, z]) tuple
• dxfattribs – dict of DXF attributes for Vertex class

append_vertices(points: Iterable[Vertex], dxfattribs: Dict[KT, VT] = None) → None
Append multiple Vertex entities at location points.

Parameters

• points – iterable of (x, y[, z]) tuples
• dxfattribs – dict of DXF attributes for Vertex class

append_formatted_vertices(points: Iterable[Vertex], format: str = 'xy', dxfattribs: Dict[KT, VT] = None) → None
Append multiple Vertex entities at location points.

Parameters

• points – iterable of (x, y, [start_width, [end_width, [bulge]]]) tuple
• format – format string, default is 'xy', see: User Defined Point Format Codes
• dxfattribs – dict of DXF attributes for Vertex class

insert_vertices(pos: int, points: Iterable[Vertex], dxfattribs: dict = None) → None
Insert vertices points into Polyline.vertices list at insertion location pos.

Parameters

• pos – insertion position of list Polyline.vertices
• points – list of (x, y[, z]) tuples
• dxfattribs – dict of DXF attributes for Vertex class

transform(m: Matrix44) → Polyline
Transform the POLYLINE entity by transformation matrix m inplace.
virtual_entities() → Iterable[Union[Line, Arc]]
Yields ‘virtual’ parts of POLYLINE as LINE, ARC or 3DFACE primitives.
This entities are located at the original positions, but are not stored in the entity database, have no handle
and are not assigned to any layout.

explode (target_layout: BaseLayout = None) → EntityQuery
Explode POLYLINE as DXF LINE, ARC or 3DFACE primitives into target layout, if the target layout
is None, the target layout is the layout of the POLYLINE entity. Returns an EntityQuery container
including all DXF primitives.

Parameters

• target_layout – target layout for DXF primitives, None for same
• as source entity. (layout) –

Vertex

A VERTEX (VERTEX DXF Reference) represents a polyline/mesh vertex.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'VERTEX'</td>
</tr>
<tr>
<td>Factory function</td>
<td>Polyline.append_vertex()</td>
</tr>
<tr>
<td>Factory function</td>
<td>Polyline.extend()</td>
</tr>
<tr>
<td>Factory function</td>
<td>Polyline.insert_vertices()</td>
</tr>
<tr>
<td>Inherited DXF Attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
</tbody>
</table>

class ezdxf.entities.Vertex

dxf.location
Vertex location (2D/3D Point OCS when 2D, WCS when 3D)

dxf.start_width
Line segment start width (float); default = 0

dxf.end_width
Line segment end width (float); default = 0

dxf.bulge
Bulge value (float); default = 0.
The bulge value is used to create arc shaped line segments.

dxf.flags
Constants defined in ezdxf.ellip.const:

<table>
<thead>
<tr>
<th>Vertex.dxf.flags</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VTX_EXTRA_VERTEX</td>
<td>X Cretd</td>
<td>vertex created by curve-fitting</td>
</tr>
<tr>
<td>VTX_CURVE_FIT_TAN</td>
<td>T Cretd</td>
<td>curve-fit tangent defined for this vertex. A curve-fit tangent direction of 0 may be omitted from the DXF output, but is significant if this bit is set.</td>
</tr>
<tr>
<td>VTX_SPLINE_VERTEX</td>
<td>X Cretd</td>
<td>vertex created by spline-fitting</td>
</tr>
<tr>
<td>VTX_SPLINE_FRAME</td>
<td>X Frame</td>
<td>Control point</td>
</tr>
<tr>
<td>VTX_3D_POLYLINE</td>
<td>X Pol</td>
<td>3D polyline vertex</td>
</tr>
<tr>
<td>VTX_3D_POLYGON_MSH</td>
<td>X Poly</td>
<td>Poly mesh</td>
</tr>
<tr>
<td>VTX_3D_POLYFACE_MSH</td>
<td>X Pol</td>
<td>Poly mesh</td>
</tr>
</tbody>
</table>
dxf.tangent
Curve fit tangent direction (float), used for 2D spline in DXF R12.

dxf.vtx1
Index of 1st vertex, if used as face (feature for experts)

dxf.vtx2
Index of 2nd vertex, if used as face (feature for experts)

dxf.vtx3
Index of 3rd vertex, if used as face (feature for experts)

dxf.vtx4
Index of 4th vertex, if used as face (feature for experts)

is_2d_polyline_vertex
is_3d_polyline_vertex
is_polygon_mesh_vertex
is_poly_face_mesh_vertex
is_face_record

format (format='xyz') → Sequence
Return formatted vertex components as tuple.

Format codes:
• “x” = x-coordinate
• “y” = y-coordinate
• “z” = z-coordinate
• “s” = start width
• “e” = end width
• “b” = bulge value
• “v” = (x, y, z) as tuple

Args: format: format string, default is “xyz”

New in version 0.14.

Polymesh

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.Polyline</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'POLYLINE'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_polymesh()</td>
</tr>
<tr>
<td>Inherited DXF Attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
</tbody>
</table>

class ezdxf.entities.Polymesh
A polymesh is a grid of m_count x n_count vertices, every vertex has its own (x, y, z) location. The Polymesh is an subclass of Polyline, DXF type is also 'POLYLINE' but get_mode() returns 'AcDbPolygonMesh'.

6.8. Reference
get_mesh_vertex(pos: Tuple[int, int]) → ezdxf.entities.polyline.DXFVertex
Get location of a single mesh vertex.
Parameters pos – 0-based (row, col) tuple, position of mesh vertex

set_mesh_vertex(pos: Tuple[int, int], point: Vertex, dxfattribs: dict = None)
Set location and DXF attributes of a single mesh vertex.
Parameters
• pos – 0-based (row, col)-tuple, position of mesh vertex
• point – (x, y, z)-tuple, new 3D coordinates of the mesh vertex
• dxfattribs – dict of DXF attributes

get_mesh_vertex_cache() → ezdxf.entities.polyline.MeshVertexCache
Get a MeshVertexCache object for this POLYMESH. The caching object provides fast access to the
location attribute of mesh vertices.

MeshVertexCache
class ezdxf.entities.MeshVertexCache
Cache mesh vertices in a dict, keys are 0-based (row, col) tuples.
Set vertex location: cache[row, col] = (x, y, z)
Get vertex location: x, y, z = cache[row, col]

vertices Dict of mesh vertices, keys are 0-based (row, col) tuples.
__getitem__(pos: Tuple[int, int]) → Vertex
Get mesh vertex location as (x, y, z)-tuple.
Parameters pos – 0-based (row, col)-tuple.

__setitem__(pos: Tuple[int, int], location: Vertex) → None
Get mesh vertex location as (x, y, z)-tuple.
Parameters
• pos – 0-based (row, col)-tuple.
• location – (x, y, z)-tuple

Polyface

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.Polyline</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'POLYLINE'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_polyface()</td>
</tr>
<tr>
<td>Inherited DXF Attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
</tbody>
</table>

See also:
Tutorial for Polyface
class ezdxf.entities.Polyface
A polyface consist of multiple location independent 3D areas called faces. The Polyface is a subclass of
Polyline, DXF type is also 'POLYLINE' but get_mode() returns 'AcDbPolyFaceMesh'.

266 Chapter 6. Contents
**append_face** *(face: FaceType, dxfattribs: Dict[KT, VT] = None) → None*

 Append a single face. A *face* is a list of \((x, y, z)\) tuples.

**Parameters**

- **face** – List\([(x, y, z)\] tuples]
- **dxfattribs** – dict of DXF attributes for *Vertex* entity

**append_faces** *(faces: Iterable[FaceType], dxfattribs: Dict[KT, VT] = None) → None*

 Append multiple *faces*. *faces* is a list of single faces and a single face is a list of \((x, y, z)\) tuples.

**Parameters**

- **faces** – list of List\([(x, y, z)\] tuples]
- **dxfattribs** – dict of DXF attributes for *Vertex* entity

**faces** () → Iterable[List[Vertex]]

 Iterable of all faces, a face is a tuple of vertices.

**Returns**  [vertex, vertex, vertex, [vertex,] face_record]

**Return type** list

**optimize** *(precision: int = 6) → None*

 Rebuilds *Polyface* including vertex optimization by merging vertices with nearly same vertex locations.

**Parameters**

- **precision** – floating point precision for determining identical vertex locations

---

**Ray**

RAY entity *(DXF Reference)* starts at Ray.dxf.point and continues to infinity (construction line).

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.XLine</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'RAY'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_ray()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

**class ezdxf.entities.Ray**

**dxf.start**

 Start point as (3D Point in WCS)

**dxf.unit_vector**

 Unit direction vector as (3D Point in WCS)

**transform** *(m: Matrix44) → Ray*

 Transform the XLINE/RAY entity by transformation matrix \(m\) inplace.

**translate** *(dx: float, dy: float, dz: float) → Ray*

 Optimized XLINE/RAY translation about \(dx\) in x-axis, \(dy\) in y-axis and \(dz\) in z-axis.
Region

REGION (DXF Reference) created by an ACIS based geometry kernel provided by the Spatial Corp. ezdxf will never interpret ACIS source code, don’t ask me for this feature.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'REGION'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_region()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.Region
    Same attributes and methods as parent class Body.

Shape

SHAPES (DXF Reference) are objects that are used like block references, each SHAPE reference can be scaled and rotated individually. The SHAPE definitions are stored in external shape files (*.SHX), and ezdxf can not create this shape files.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'SHAPE'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_shape()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
</tbody>
</table>

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.Shape

    dxf.insert
        Insertion location as (2D/3D Point in WCS)
    dxf.name
        Shape name (str)
    dxf.size
        Shape size (float)
    dxf.rotation
        Rotation angle in degrees; default value is 0
    dxf.xscale
        Relative X scale factor (float); default value is 1
    dxf.oblique
        Oblique angle in degrees (float); default value is 0
**transform** \((m: \text{Matrix44}) \rightarrow \text{Shape}\)

Transform the SHAPE entity by transformation matrix \(m\) inplace.

**Solid**

SOLID (DXF Reference) is a filled triangle or quadrilateral. Access vertices by name \((\text{entity}.\text{dxf}\.\text{vtx}0 = (1.7, 2.3))\) or by index \((\text{entity}[0] = (1.7, 2.3))\).

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>\text{ezdxf.entities.DXFGraphic}</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'SOLID'</td>
</tr>
<tr>
<td>Factory function</td>
<td>\text{ezdxf.layouts.BaseLayout.add_solid()}</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td><em>Common graphical DXF attributes</em></td>
</tr>
</tbody>
</table>

**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!

```python
class \text{ezdxf.entities.Solid}
```

dxf.\text{vtx}0
Location of 1. vertex (2D/3D Point in \(OCS\))

dxf.\text{vtx}1
Location of 2. vertex (2D/3D Point in \(OCS\))

dxf.\text{vtx}2
Location of 3. vertex (2D/3D Point in \(OCS\))

dxf.\text{vtx}3
Location of 4. vertex (2D/3D Point in \(OCS\))

**transform** \((m: \text{Matrix44}) \rightarrow \text{Solid}\)

Transform the SOLID/TRACE entity by transformation matrix \(m\) inplace.

**vertices** \((\text{close: bool=False}) \rightarrow \text{List[Vec3]}\)

Returns OCS vertices in correct order, if argument \(\text{close}\) is \text{True}, last vertex == first vertex. Does **not** return duplicated last vertex if represents a triangle.

New in version 0.15.

**wcs_vertices** \((\text{close: bool=False}) \rightarrow \text{List[Vec3]}\)

Returns WCS vertices in correct order, if argument \(\text{close}\) is \text{True}, last vertex == first vertex. Does **not** return duplicated last vertex if represents a triangle.

New in version 0.15.

**Spline**

SPLINE curve (DXF Reference), all coordinates have to be 3D coordinates even the spline is only a 2D planar curve.

The spline curve is defined by control points, knot values and weights. The control points establish the spline, the various types of knot vector determines the shape of the curve and the weights of rational splines define how strong a control point influences the shape.

To create a Spline curve you just need a bunch of fit points - knot values and weights are optional (tested with AutoCAD 2010). If you add additional data, be sure that you know what you do.
New in version 0.16: The function `ezdxf.math.fit_points_to_cad_cv()` calculates control vertices from given fit points. This control vertices define a cubic B-spline which matches visually the SPLINE entities created by BricsCAD and AutoCAD from fit points.

See also:

- [Wikipedia article about B_splines](#)
- [Department of Computer Science and Technology at the Cambridge University](#)
- [Tutorial for Spline](#)

### Factory Functions

<table>
<thead>
<tr>
<th>Subclass of</th>
<th><code>ezdxf.entities.DXFGraphic</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'SPLINE'</td>
</tr>
<tr>
<td>Factory function</td>
<td>see table below</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td><code>Common graphical DXF attributes</code></td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

#### class `ezdxf.entities.Spline`
All points in WCS as (x, y, z) tuples

- **dxf.degree**
  Degree of the spline curve (int).

- **dxf.flags**
  Bit coded option flags, constants defined in `ezdxf.lldxf.const`:

<table>
<thead>
<tr>
<th>dxf.flags</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSED_SPLINE</td>
<td>1</td>
<td>Spline is closed</td>
</tr>
<tr>
<td>PERIODIC_SPLINE</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RATIONAL_SPLINE</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PLANAR_SPLINE</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>LINEAR_SPLINE</td>
<td>16</td>
<td>planar bit is also set</td>
</tr>
</tbody>
</table>

- **dxf.n_knots**
  Count of knot values (int), automatically set by `ezdxf` (read only)

- **dxf.n_fit_points**
  Count of fit points (int), automatically set by `ezdxf` (read only)

- **dxf.n_control_points**
  Count of control points (int), automatically set by `ezdxf` (read only)

- **dxf.knot_tolerance**
  Knot tolerance (float); default = $1e-10$
dx.f.fit_tolerance
    Fit tolerance (float); default = 1e-10

dx.f.control_point_tolerance
    Control point tolerance (float); default = 1e-10

dx.f.start_tangent
    Start tangent vector as (3D vector in WCS)

dx.f.end_tangent
    End tangent vector as (3D vector in WCS)

closed
    True if spline is closed. A closed spline has a connection from the last control point to the first control point. (read/write)

control_points
    VertexArray of control points in WCS.

fit_points
    VertexArray of fit points in WCS.

knots
    Knot values as array.array('d').

weights
    Control point weights as array.array('d').

control_point_count () → int
    Count of control points.

fit_point_count () → int
    Count of fit points.

knot_count () → int
    Count of knot values.

construction_tool() → BSpline
    Returns the construction tool ezdxf.math.BSpline.

apply_construction_tool (s: BSpline) → Spline
    Apply SPLINE data from a BSpline construction tool or from a geomdl.BSpline.Curve object.

flattening (distance: float, segments: int = 4) → Iterable[Vec3]
    Adaptive recursive flattening. The argument segments is the minimum count of approximation segments between two knots, if the distance from the center of the approximation segment to the curve is bigger than distance the segment will be subdivided.

    Parameters

    • distance – maximum distance from the projected curve point onto the segment chord.
    • segments – minimum segment count between two knots

    New in version 0.15.

set_open_uniform (control_points: Sequence[Vertex], degree: int = 3) → None
    Open B-spline with uniform knot vector, start and end at your first and last control points.

set_uniform (control_points: Sequence[Vertex], degree: int = 3) → None
    B-spline with uniform knot vector, does NOT start and end at your first and last control points.

set_closed (control_points: Sequence[Vertex], degree=3) → None
    Closed B-spline with uniform knot vector, start and end at your first control point.
ezdxf Documentation, Release 0.16.2

**set_open_rational** *(control_points: Sequence[Vertex], weights: Sequence[float], degree: int = 3) → None*

Open rational B-spline with uniform knot vector, start and end at your first and last control points, and has additional control possibilities by weighting each control point.

**set_uniform_rational** *(control_points: Sequence[Vertex], weights: Sequence[float], degree: int = 3) → None*

Rational B-spline with uniform knot vector, does NOT start and end at your first and last control points, and has additional control possibilities by weighting each control point.

**set_closed_rational** *(control_points: Sequence[Vertex], weights: Sequence[float], degree: int = 3) → None*

Closed rational B-spline with uniform knot vector, start and end at your first control point, and has additional control possibilities by weighting each control point.

**transform** *(m: Matrix44) → Spline*

Transform the SPLINE entity by transformation matrix \( m \) inplace.

**classmethod from_arc** *(entity: DXFGraphic) → Spline*

Create a new SPLINE entity from a CIRCLE, ARC or ELLIPSE entity.

The new SPLINE entity has no owner, no handle, is not stored in the entity database nor assigned to any layout!

**Surface**

SURFACE (DXF Reference) created by an ACIS based geometry kernel provided by the Spatial Corp. ezdxf will never interpret ACIS source code, don’t ask me for this feature.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'SURFACE'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_surface()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.Surface

Same attributes and methods as parent class Body.

```python

dxf.u_count

Number of U isolines.

dxf.v_count

Number of V2 isolines.
```

**ExtrudedSurface**

(DXF Reference)
class ezdxf.entities.ExtrudedSurface

Same attributes and methods as parent class Surface.

dxf.class_id

dxf.sweep_vector

dxf.draft_angle

dxf.draft_start_distance

dxf.draft_end_distance

dxf.twist_angle

dxf.scale_factor

dxf.align_angle

dxf.solid

<table>
<thead>
<tr>
<th>dxf.sweep_alignment_flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

dxf.align_start

dxf.bank

dxf.base_point_set

dxf.sweep_entity_transform_computed

dxf.path_entity_transform_computed

dxf.reference_vector_for_controlling_twist

transformation_matrix_extruded_entity
type: Matrix44

sweep_entity_transformation_matrix
type: Matrix44

path_entity_transformation_matrix
type: Matrix44
Subclass of | `ezdxf.entities.Surface`
--- | ---
DXF type | 'LOFTEDSURFACE'
Factory function | `ezdxf.layouts.BaseLayout.add_lofted_surface()`
Inherited DXF attributes | Common graphical DXF attributes
Required DXF version | DXF R2007 ('AC1021')

**class** `ezdxf.entities.LoftedSurface`

Same attributes and methods as parent class `Surface`.

- `dxf.plane_normal_lofting_type`
- `dxf.start_draft_angle`
- `dxf.end_draft_angle`
- `dxf.start_draft_magnitude`
- `dxf.end_draft_magnitude`
- `dxf.arc_length_parameterization`
- `dxf.no_twist`
- `dxf.align_direction`
- `dxf.simple_surfaces`
- `dxf.closed_surfaces`
- `dxf.solid`
- `dxf.ruled_surface`
- `dxf.virtual_guide`

`set_transformation_matrix_lofted_entity`<br>type: `Matrix44`

**RevolvedSurface**

(DXF Reference)

Subclass of | `ezdxf.entities.Surface`
--- | ---
DXF type | 'REVOLVEDSURFACE'
Factory function | `ezdxf.layouts.BaseLayout.add_revolved_surface()`
Inherited DXF attributes | Common graphical DXF attributes
Required DXF version | DXF R2007 ('AC1021')

**class** `ezdxf.entities.RevolvedSurface`

Same attributes and methods as parent class `Surface`.

- `dxf.class_id`
- `dxf.axis_point`
- `dxf.axis_vector`
- `dxf.revolve_angle`
- `dxf.start_angle`
SweptSurface

**ezdxf Documentation, Release 0.16.2**

```python
dxf.draft_angle
dxf.start_draft_distance
dxf.end_draft_distance
dxf.twist_angle
dxf.solid
dxf.close_to_axis

transformation_matrix_revolved_entity
type: Matrix44
```

**SweptSurface**

((DXF Reference)

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.Surface</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'SWEPTSURFACE'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_swept_surface()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2007 ('AC1021')</td>
</tr>
</tbody>
</table>

```python
class ezdxf.entities.SweptSurface
    Same attributes and methods as parent class Surface.
dxf.swept_entity_id
dxf.path_entity_id
dxf.draft_angle
draft_start_distance
        dxf.draft_end_distance
        dxf.twist_angle
dxf.scale_factor
dxf.align_angle
dxf.solid
dxf.sweep_alignment
dxf.align_start
dxf.bank
dxf.base_point_set
        dxf.sweep_entity_transform_computed
dxf.path_entity_transform_computed
dxf.reference_vector_for_controlling_twist
        transformation_matrix_sweep_entity
type: Matrix44
```

6.8. Reference
transformation_matrix_path_entity()
  type: Matrix44

sweep_entity_transformation_matrix()
  type: Matrix44

path_entity_transformation_matrix()
  type: Matrix44

Text

The single line TEXT entity (DXF Reference). The style attribute stores the associated Textstyle entity as string, which defines the basic font properties. The text size is stored as cap height in the height attribute in drawing units.

See also:
See the documentation for the Textstyle class to understand the limitations of text representation in the DXF format.

Tutorial for Text

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'TEXT'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_text()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
</tbody>
</table>

**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.Text

dxf.text
  Text content as string.

dxf.insert
  First alignment point of text (2D/3D Point in OCS), relevant for the adjustments “LEFT”, “ALIGNED” and “FIT”.

dxf.align_point
  The main alignment point of text (2D/3D Point in OCS), if the alignment is anything else than “LEFT”, or the second alignment point for the “ALIGNED” and “FIT” alignments.

dxf.height
  Text height in drawing units as float value, the default value is 1.

dxf.rotation
  Text rotation in degrees as float value, the default value is 0.

dxf.oblique
  Text oblique angle (slanting) in degrees as float value, the default value is 0 (straight vertical text).

dxf.style
  Textstyle name as case insensitive string, the default value is “Standard”

dxf.width
  Width scale factor as float value, the default value is 1.
**ezdxf Documentation, Release 0.16.2**

**dxf.halign**
Horizontal alignment flag as int value, use the `set_pos()` and `get_align()` methods to handle text alignment, the default value is 0.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Left</td>
</tr>
<tr>
<td>2</td>
<td>Right</td>
</tr>
<tr>
<td>3</td>
<td>Aligned (if vertical alignment = 0)</td>
</tr>
<tr>
<td>4</td>
<td>Middle (if vertical alignment = 0)</td>
</tr>
<tr>
<td>5</td>
<td>Fit (if vertical alignment = 0)</td>
</tr>
</tbody>
</table>

**dxf.valign**
Vertical alignment flag as int value, use the `set_pos()` and `get_align()` methods to handle text alignment, the default value is 0.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Baseline</td>
</tr>
<tr>
<td>1</td>
<td>Bottom</td>
</tr>
<tr>
<td>2</td>
<td>Middle</td>
</tr>
<tr>
<td>3</td>
<td>Top</td>
</tr>
</tbody>
</table>

**dxf.text_generation_flag**
Text generation flags as int value, use the `is_backward` and `is_upside_down` attributes to handle this flags.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>text is backward (mirrored in X)</td>
</tr>
<tr>
<td>4</td>
<td>text is upside down (mirrored in Y)</td>
</tr>
</tbody>
</table>

**set_pos** *(p1: Vertex, p2: Vertex = None, align: str = None) → Text*
Set text alignment, valid alignments are:

<table>
<thead>
<tr>
<th>Vertical</th>
<th>Left</th>
<th>Center</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top</td>
<td>TOP_LEFT</td>
<td>TOP_CENTER</td>
<td>TOP_RIGHT</td>
</tr>
<tr>
<td>Middle</td>
<td>MIDDLE_LEFT</td>
<td>MIDDLE_CENTER</td>
<td>MIDDLE_RIGHT</td>
</tr>
<tr>
<td>Bottom</td>
<td>BOTTOM_LEFT</td>
<td>BOTTOM_CENTER</td>
<td>BOTTOM_RIGHT</td>
</tr>
<tr>
<td>Baseline</td>
<td>LEFT</td>
<td>CENTER</td>
<td>RIGHT</td>
</tr>
</tbody>
</table>

Alignments “ALIGNED” and “FIT” are special, they require a second alignment point, text is aligned on the virtual line between these two points and sit vertical at the base line.

- “ALIGNED”: Text is stretched or compressed to fit exactly between `p1` and `p2` and the text height is also adjusted to preserve height/width ratio.
- “FIT”: Text is stretched or compressed to fit exactly between `p1` and `p2` but only the text width is adjusted, the text height is fixed by the `dxf.height` attribute.
- “MIDDLE”: also a special adjustment, centered text like “MIDDLE_CENTER”, but vertical centred at the total height of the text.

**Parameters**

- `p1` – first alignment point as (x, y[, z]) tuple
- `p2` – second alignment point as (x, y[, z]) tuple, required for “ALIGNED” and “FIT” else ignored
- `align` – new alignment, `None` for preserve existing alignment.
get_pos() \rightarrow Tuple[str, ezdxf.math._vector.Vec3, Optional[ezdxf.math._vector.Vec3]]

Returns a tuple \((align, p1, p2)\), \(align\) is the alignment method, \(p1\) is the alignment point, \(p2\) is only relevant if \(align\) is “ALIGNED” or “FIT”, otherwise it is \(None\).

get_align() \rightarrow str

Returns the actual text alignment as string, see also \(set_pos()\).

set_align(align: str = 'LEFT') \rightarrow Text

Just for experts: Sets the text alignment without setting the alignment points, set adjustment points \attr:dx.insert and \(dx.align_point\) manually.

Parameters align – test alignment, see also \(set_pos()\)

transform\((m: \text{Matrix44})\) \rightarrow Text

Transform the TEXT entity by transformation matrix \(m\) inplace.

translate\((dx: \text{float}, dy: \text{float}, dz: \text{float})\) \rightarrow Text

Optimized TEXT/ATTRIB/ATTDEF translation about \(dx\) in x-axis, \(dy\) in y-axis and \(dz\) in z-axis, returns \(self\).

plain_text() \rightarrow str

Returns text content without formatting codes.

font_name() \rightarrow str

Returns the font name of the associated \textit{Textstyle}.

fit_length() \rightarrow float

Returns the text length for alignments “FIT” and “ALIGNED”, defined by the distance from the insertion point to the align point or 0 for all other alignments.

Trace

TRACE entity \(\text{(DXF Reference)}\) is solid filled triangle or quadrilateral. Access vertices by name \(\text{(entity.dxf.vtx0 = (1.7, 2.3))}\) or by index \(\text{(entity[0] = (1.7, 2.3))}\). I don’t know the difference between SOLID and TRACE.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'TRACE'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_trace()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
</tbody>
</table>

**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.Trace

dxf.vtx0
Location of 1. vertex (2D/3D Point in \textit{OCS})

dxf.vtx1
Location of 2. vertex (2D/3D Point in \textit{OCS})

dxf.vtx2
Location of 3. vertex (2D/3D Point in \textit{OCS})

dxf.vtx3
Location of 4. vertex (2D/3D Point in \textit{OCS})
transform \((m: \text{Matrix44}) \rightarrow \text{Trace}\)
Transform the SOLID/TRACE entity by transformation matrix \(m\) inplace.

vertices \((\text{close: bool=False}) \rightarrow \text{List[Vec3]}\)
Returns OCS vertices in correct order, if argument close is True, last vertex == first vertex. Does not return duplicated last vertex if represents a triangle.
New in version 0.15.

wcs_vertices \((\text{close: bool=False}) \rightarrow \text{List[Vec3]}\)
Returns WCS vertices in correct order, if argument close is True, last vertex == first vertex. Does not return duplicated last vertex if represents a triangle.
New in version 0.15.

Underlay

UNDERLAY entity (DXF Reference) links an underlay file to the DXF file, the file itself is not embedded into the DXF file, it is always a separated file. The (PDF)UNDERLAY entity is like a block reference, you can use it multiple times to add the underlay on different locations with different scales and rotations. But therefore you need a also a (PDF)DEFINITION entity, see UnderlayDefinition.

The DXF standard supports three different file formats: PDF, DWF (DWFx) and DGN. An Underlay can be clipped by a rectangle or a polygon path. The clipping coordinates are 2D OCS coordinates in drawing units but without scaling.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>internal base class</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_underlay()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

class ezdxf.entities.Underlay
Base class of PdfUnderlay, DwfUnderlay and DgnUnderlay

dxf.insert
Insertion point, lower left corner of the image in OCS.

dxf.scale_x
Scaling factor in x-direction (float)

dxf.scale_y
Scaling factor in y-direction (float)

dxf.scale_z
Scaling factor in z-direction (float)

dxf.rotation
ccw rotation in degrees around the extrusion vector (float)

dxf.extrusion
extrusion vector, default = \((0, 0, 1)\)

dxf.underlay_def_handle
Handle to the underlay definition entity, see UnderlayDefinition

dxf.flags
<table>
<thead>
<tr>
<th>dxf.flags</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNDERLAY_CLIPPING</td>
<td>1</td>
<td>clipping is on/off</td>
</tr>
<tr>
<td>UNDERLAY_ON</td>
<td>2</td>
<td>underlay is on/off</td>
</tr>
<tr>
<td>UNDERLAY_MONOCHROME</td>
<td>4</td>
<td>Monochrome</td>
</tr>
<tr>
<td>UNDERLAY_ADJUST_FOR_BACKGROUND</td>
<td>8</td>
<td>Adjust for background</td>
</tr>
</tbody>
</table>

dxf.\_contrast
Contrast value (20 - 100; default = 100)

dxf.\_fade
Fade value (0 - 80; default = 0)

clipping
True or False (read/write)

on
True or False (read/write)

monochrome
True or False (read/write)

adjust\_for\_background
True or False (read/write)

scale
Scaling (x, y, z) tuple (read/write)

boundary\_path
Boundary path as list of vertices (read/write).

Two vertices describe a rectangle (lower left and upper right corner), more than two vertices is a polygon as clipping path.

get\_underlay\_def() → UnderlayDefinition
Returns the associated DEFINITION entity. see UnderlayDefinition.

set\_underlay\_def(underlay\_def: UnderlayDefinition) → None
Set the associated DEFINITION entity. see UnderlayDefinition.

reset\_boundary\_path()
Removes the clipping path.

PdfUnderlay

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.Underlay</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'PDFUNDERLAY'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_underlay()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

class ezdxf.entities.PdfUnderlay
PDF underlay.
DwfUnderlay

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.Underlay</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'DWFUNDERLAY'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_underlay()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

class ezdxf.entities.DwfUnderlay

DWF underlay.

DgnUnderlay

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.Underlay</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'DGNUNDERLAY'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_underlay()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

class ezdxf.entities.DgnUnderlay

DGN underlay.

Viewport

The VIEWPORT (DXF Reference) entity is a window from a paperspace layout to the modelspace.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'VIEWPORT'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.Paperspace.add_viewport()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
</tbody>
</table>

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class ezdxf.entities.Viewport

dxf.center
    Center point of the viewport located in the paper space layout in paper space units stored as 3D point.
    (Error in the DXF reference)

dxf.width
    Viewport width in paperspace units (float)

dxf.height
    Viewport height in paperspace units (float)

dxf.status
    Viewport status field (int)
On, but is fully off screen, or is one of the viewports that is not active because the $MAXACTVP count is currently being exceeded.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Off</td>
</tr>
<tr>
<td>&gt;0</td>
<td>On and active. The value indicates the order of stacking for the viewports, where 1 is the active viewport, 2 is the next, and so forth</td>
</tr>
</tbody>
</table>

**dxf.id**  
Viewport id (int)

**dxf.view_center_point**  
View center point in modelspace stored as 2D point, but represents a WCS point. (Error in the DXF reference)

**dxf.snap_base_point**

**dxf.snap_spacing**

**dxf.snap_angle**

**dxf.grid_spacing**

**dxf.view_direction_vector**  
View direction (3D vector in WCS).

**dxf.view_target_point**  
View target point (3D point in WCS).

**dxf.perspective_lens_length**  
Lens focal length in mm as 35mm film equivalent.

**dxf.front_clip_plane_z_value**

**dxf.back_clip_plane_z_value**

**dxf.view_height**  
View height in WCS.

**dxf.view_twist_angle**

**dxf.circle_zoom**

**dxf.flags**  
Viewport status bit-coded flags:
<table>
<thead>
<tr>
<th>Value (in hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (0x1)</td>
<td>Enables perspective mode</td>
</tr>
<tr>
<td>2 (0x2)</td>
<td>Enables front clipping</td>
</tr>
<tr>
<td>4 (0x4)</td>
<td>Enables back clipping</td>
</tr>
<tr>
<td>8 (0x8)</td>
<td>Enables UCS follow</td>
</tr>
<tr>
<td>16 (0x10)</td>
<td>Enables front clip not at eye</td>
</tr>
<tr>
<td>32 (0x20)</td>
<td>Enables UCS icon visibility</td>
</tr>
<tr>
<td>64 (0x40)</td>
<td>Enables UCS icon at origin</td>
</tr>
<tr>
<td>128 (0x80)</td>
<td>Enables fast zoom</td>
</tr>
<tr>
<td>256 (0x100)</td>
<td>Enables snap mode</td>
</tr>
<tr>
<td>512 (0x200)</td>
<td>Enables grid mode</td>
</tr>
<tr>
<td>1024 (0x400)</td>
<td>Enables isometric snap style</td>
</tr>
<tr>
<td>2048 (0x800)</td>
<td>Enables hide plot mode</td>
</tr>
<tr>
<td>4096 (0x1000)</td>
<td>klIsoPairTop. If set and klIsoPairRight is not set, then isopair top is enabled. If both klIsoPairTop and klIsoPairRight are set, then isopair left is enabled</td>
</tr>
<tr>
<td>8192 (0x2000)</td>
<td>klIsoPairRight. If set and klIsoPairTop is not set, then isopair right is enabled</td>
</tr>
<tr>
<td>16384 (0x4000)</td>
<td>Enables viewport zoom locking</td>
</tr>
<tr>
<td>32768 (0x8000)</td>
<td>Currently always enabled</td>
</tr>
<tr>
<td>65536 (0x10000)</td>
<td>Enables non-rectangular clipping</td>
</tr>
<tr>
<td>131072 (0x20000)</td>
<td>Turns the viewport off</td>
</tr>
<tr>
<td>262144 (0x40000)</td>
<td>Enables the display of the grid beyond the drawing limits</td>
</tr>
<tr>
<td>524288 (0x80000)</td>
<td>Enable adaptive grid display</td>
</tr>
<tr>
<td>1048576 (0x100000)</td>
<td>Enables subdivision of the grid below the set grid spacing when the grid display is adaptive</td>
</tr>
<tr>
<td>2097152 (0x200000)</td>
<td>Enables grid follows workplane switching</td>
</tr>
</tbody>
</table>

**dxf.clipping_boundary_handle**

**dxf.plot_style_name**

**dxf.render_mode**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2D Optimized (classic 2D)</td>
</tr>
<tr>
<td>1</td>
<td>Wireframe</td>
</tr>
<tr>
<td>2</td>
<td>Hidden line</td>
</tr>
<tr>
<td>3</td>
<td>Flat shaded</td>
</tr>
<tr>
<td>4</td>
<td>Gouraud shaded</td>
</tr>
<tr>
<td>5</td>
<td>Flat shaded with wireframe</td>
</tr>
<tr>
<td>6</td>
<td>Gouraud shaded with wireframe</td>
</tr>
</tbody>
</table>

**dxf.ucs_per_viewport**

**dxf.ucs_icon**
**ezdxf Documentation, Release 0.16.2**

- **dx.ucs_origin**
  UCS origin as 3D point.

- **dx.ucs_x_axis**
  UCS x-axis as 3D vector.

- **dx.ucs_y_axis**
  UCS y-axis as 3D vector.

- **dx.ucs_handle**
  Handle of **UCSTable** if UCS is a named UCS. If not present, then UCS is unnamed.

- **dx.ucs_ortho_type**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>not orthographic</td>
</tr>
<tr>
<td>1</td>
<td>Top</td>
</tr>
<tr>
<td>2</td>
<td>Bottom</td>
</tr>
<tr>
<td>3</td>
<td>Front</td>
</tr>
<tr>
<td>4</td>
<td>Back</td>
</tr>
<tr>
<td>5</td>
<td>Left</td>
</tr>
<tr>
<td>6</td>
<td>Right</td>
</tr>
</tbody>
</table>

- **dx.ucs_base_handle**
  Handle of **UCSTable** of base UCS if UCS is orthographic (**Viewport.dx.ucs_ortho_type** is non-zero). If not present and **Viewport.dx.ucs_ortho_type** is non-zero, then base UCS is taken to be WORLD.

- **dx.elevation**

- **dx.shade_plot_mode**
  (DXF R2004)

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>As Displayed</td>
</tr>
<tr>
<td>1</td>
<td>Wireframe</td>
</tr>
<tr>
<td>2</td>
<td>Hidden</td>
</tr>
<tr>
<td>3</td>
<td>Rendered</td>
</tr>
</tbody>
</table>

- **dx.grid_frequency**
  Frequency of major grid lines compared to minor grid lines. (DXF R2007)

- **dx.background_handle**

- **dx.shade_plot_handle**

- **dx.visual_style_handle**

- **dx.default_lighting_flag**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>One distant light</td>
</tr>
<tr>
<td>1</td>
<td>Two distant lights</td>
</tr>
</tbody>
</table>

- **dx.view_brightness**

- **dx.view_contrast**
ezdxf Documentation, Release 0.16.2

**Wipeout**

THE WIPEOUT (DXF Reference) entity is a polygonal area that masks underlying objects with the current background color. The WIPEOUT entity is based on the IMAGE entity, but usage does not require any knowledge about the IMAGE entity.

The handles to the support entities *ImageDef* and *ImageDefReactor* are always “0”, both are not needed by the WIPEOUT entity.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'WIPEOUT'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_wipeout()</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>Common graphical DXF attributes</td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!

```python
class ezdxf.entities.Wipeout

    set_masking_area (vertices: Iterable[Vertex]) -> None

    Set a new masking area, the area is placed in the layout xy-plane.
```

**XLine**

XLINE entity (DXF Reference) is a construction line that extents to infinity in both directions.

```python
Subclass of                  | ezdxf.entities.DXFGraphic
DXF type                     | 'XLINE'
Factory function             | ezdxf.layouts.BaseLayout.add_xline()
Inherited DXF attributes     | Common graphical DXF attributes
Required DXF version          | DXF R2000 ('AC1015')
```
class ezdxf.entities.XLine

    dxf.start
    Location point of line as (3D Point in WCS)
    dxf.unit_vector
    Unit direction vector as (3D Point in WCS)

    transform (m: Matrix44) → XLine
    Transform the XLINE/RAY entity by transformation matrix m inplace.

    translate (dx: float, dy: float, dz: float) → XLine
    Optimized XLINE/RAY translation about dx in x-axis, dy in y-axis and dz in z-axis.

DXF Objects

DXFObject

Common base class for all non-graphical DXF objects.

class ezdxf.entities.DXFObject
    Subclass of ezdxf.entities.DXFEntity

Dictionary

The DICTIONARY is a general storage entity.

AutoCAD maintains items such as MLINE_STYLES and GROUP definitions as objects in dictionaries. Other applications are free to create and use their own dictionaries as they see fit. The prefix 'ACAD_' is reserved for use by AutoCAD applications.

Dictionary entries are (key, DXFEntity) pairs. At loading time the value could be a str, because at this time not all objects are already stored in the EntityDB, and have to be acquired later.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'DICTIONARY'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.sections.objects.ObjectsSection.add_dictionary()</td>
</tr>
</tbody>
</table>

Warning: Do not instantiate object classes by yourself - always use the provided factory functions!

class ezdxf.entities.Dictionary

    dxf.hard_owned
    If set to 1, indicates that elements of the dictionary are to be treated as hard-owned.

    dxf cloning
    Duplicate record cloning flag (determines how to merge duplicate entries, ignored by ezdxf):
**is_hard_owner**
Returns True if `Dictionary` is hard owner of entities. Hard owned entities will be destroyed by deleting the dictionary.

__len__() → int
Returns count of items.

__contains__(key: str) → bool
Returns True if key exist.

__getitem__(key: str) → DXFEntity
Return the value for key, raises a DXFKeyError if key does not exist.

__setitem__(key: str, value: DXFEntity) → None
Add item as (key, value) pair to dictionary.

__delitem__(key: str) → None
Delete entry key from the dictionary, raises DXFKeyError if key does not exist.

keys() → KeysView
Returns KeysView of all dictionary keys.

items() → ItemsView
Returns ItemsView for all dictionary entries as (key, DXFEntity) pairs.

count() → int
Returns count of items.

get(key: str, default: Any = DXFKeyError) → DXFEntity
Returns DXFEntity for key, if key exist, else default or raises a DXFKeyError for default = DXFKeyError.

add(key: str, value: DXFEntity) → None
Add entry (key, value).

remove(key: str) → None
Delete entry key. Raises DXFKeyError, if key does not exist. Deletes also hard owned DXF objects from OBJECTS section.

discard(key: str) → None
Delete entry key if exists. Does NOT raise an exception if key not exist and does not delete hard owned DXF objects.

clear() → None
Delete all entries from Dictionary, deletes hard owned DXF objects from OBJECTS section.

add_new_dict(key: str, hard_owned: bool = False) → Dictionary
Create a new sub Dictionary.

Parameters
- **key** – name of the sub dictionary
- **hard_owned** – entries of the new dictionary are hard owned
**get_required_dict** (*key: str*) → Dictionary
Get entry *key* or create a new *Dictionary*, if *Key* not exist.

**add_dict_var** (*key: str, value: str*) → DictionaryVar
Add new DictionaryVar.

Parameters
- **key** – entry name as string
- **value** – entry value as string

---

**DictionaryWithDefault**

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'ACDBDICTIONARYWDFLT'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.sections.objects.ObjectsSection.add_dictionary_with_default()</td>
</tr>
</tbody>
</table>

```python
class ezdxf.entities.DictionaryWithDefault

dxf.default
   Handle to default entry as hex string like FF00.

get (*key: str*) → DXFEntity
   Returns DXFEntity for *key* or the predefined dictionary wide dxf.default entity if *key* does not exist or None if default value also not exist.

set_default (*default: ezdxf.entities.dxfentity.DXFEntity*) → None
   Set dictionary wide default entry.

Parameters default – default entry as DXFEntity
```

---

**DictionaryVar**

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'DICTIONARYVAR'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.entities.Dictionary.add_dict_var()</td>
</tr>
</tbody>
</table>

```python
dxf.schema
   Object schema number (currently set to 0)

dxf.value
   Value as string.
```

---

**GeoData**

The GEODATA entity is associated to the Modelspace object.

```python
<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'GEODATA'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.Modelspace.new_geodata()</td>
</tr>
<tr>
<td>Required DXF version</td>
<td>R2010 ('AC1024')</td>
</tr>
</tbody>
</table>
```
class ezdxf.entities.GeoData

    dxf.version

    +----+----+
    | 1  | R2009 |
    | 2  | R2010 |
    +----+----+

    dxf.coordinate_type

    +----+------------------+
    | 0   | unknown          |
    | 1   | local grid       |
    | 2   | projected grid   |
    | 3   | geographic (latitude/longitude) |
    +----+------------------+

    dxf.block_record_handle
    Handle of host BLOCK_RECORD table entry, in general the Modelspace.

    Changed in version 0.10: renamed from dxf.block_record

    dxf.design_point
    Reference point in WCS coordinates.

    dxf.reference_point
    Reference point in geo coordinates, valid only when coordinate type is local grid. The difference between dxf.design_point and dxf.reference_point defines the translation from WCS coordinates to geo-coordinates.

    dxf.north_direction
    North direction as 2D vector. Defines the rotation (about the dxf.design_point) to transform from WCS coordinates to geo-coordinates

    dxf.horizontal_unit_scale
    Horizontal unit scale, factor which converts horizontal design coordinates to meters by multiplication.

    dxf.vertical_unit_scale
    Vertical unit scale, factor which converts vertical design coordinates to meters by multiplication.

    dxf.horizontal_units
    Horizontal units (see BlockRecord). Will be 0 (Unitless) if units specified by horizontal unit scale is not supported by AutoCAD enumeration.

    dxf.vertical_units
    Vertical units (see BlockRecord). Will be 0 (Unitless) if units specified by vertical unit scale is not supported by AutoCAD enumeration.

    dxf.up_direction
    Up direction as 3D vector.
**ezdxf Documentation, Release 0.16.2**

--

**dxfscale_estimation_method**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>none</td>
</tr>
<tr>
<td>2</td>
<td>user specified scale factor</td>
</tr>
<tr>
<td>3</td>
<td>grid scale at reference point</td>
</tr>
<tr>
<td>4</td>
<td>prismatic</td>
</tr>
</tbody>
</table>

**dxf.sea_level_correction**

Bool flag specifying whether to do sea level correction.

**dxf.user_scale_factor**

**dxf.sea_level_elevation**

**dxf.coordinate_projection_radius**

**dxf.geo_rss_tag**

**dxf.observation_from_tag**

**dxf.observation_to_tag**

**dxf.mesh_faces_count**

**source_vertices**

2D source vertices in the CRS of the GeoData as `VertexArray`. Used together with `target_vertices` to define the transformation from the CRS of the GeoData to WGS84.

**target_vertices**

2D target vertices in WGS84 (EPSG:4326) as `VertexArray`. Used together with `source_vertices` to define the transformation from the CRS of the geoData to WGS84.

**faces**

List of face definition tuples, each face entry is a 3-tuple of vertex indices (0-based).

**coordinate_system_definition**

The coordinate system definition string. Stored as XML. Defines the CRS used by the GeoData. The EPSG number and other details like the axis-ordering of the CRS is stored.

**get_crs() → Tuple[int, bool]**

Returns the EPSG index and axis-ordering, axis-ordering is `True` if first axis is labeled “E” or “W” and `False` if first axis is labeled “N” or “S”.

If axis-ordering is `False` the CRS is not compatible with the `__geo_interface__` or GeoJSON (see chapter 3.1.1).

**Raises** `InvalidGeoDataException` – for invalid or unknown XML data

The EPSG number is stored in a tag like:

```xml
<Alias id="27700" type="CoordinateSystem">
  <ObjectId>OSGB1936.NationalGrid</ObjectId>
  <Namespace>EPSG Code</Namespace>
</Alias>
```

The axis-ordering is stored in a tag like:

```xml
<Axis uom="METER">
  <CoordinateSystemAxis>
    <AxisOrder>1</AxisOrder>
  </CoordinateSystemAxis>
</Axis>
```

(continues on next page)
<AxisName>Easting</AxisName>
<AxisAbbreviation>E</AxisAbbreviation>
<AxisDirection>east</AxisDirection>
</CoordinateSystemAxis>
<CoordinateSystemAxis>
<AxisOrder>2</AxisOrder>
<AxisName>Northing</AxisName>
<AxisAbbreviation>N</AxisAbbreviation>
<AxisDirection>north</AxisDirection>
</CoordinateSystemAxis>
</Axis>

get_crs_transformation(no_checks: bool = False) \(\rightarrow\) Tuple[Matrix44, int]

Returns the transformation matrix and the EPSG index to transform WCS coordinates into CRS coordinates. Because of the lack of proper documentation this method works only for tested configurations, set argument no_checks to True to use the method for untested geodata configurations, but the results may be incorrect.

Supports only “Local Grid” transformation!

Raises InvalidGeoDataException – for untested geodata configurations

setup_local_grid(design_point: Vec3, reference_point: Vec3, north_direction: Vec2=Y_AXIS, crs: str=EPSG_3395)

Setup local grid coordinate system. This method is designed to setup CRS similar to EPSG:3395 World Mercator, the basic features of the CRS should fulfill this assumptions:

- base unit of reference coordinates is 1 meter
- right-handed coordinate system: +Y=north/+X=east/+Z=up

The CRS string is not validated nor interpreted!

Hint: The reference point must be a 2D cartesian map coordinate and not a globe (lon/lat) coordinate like stored in GeoJSON or GPS data.

Parameters

- **design_point** – WCS coordinates of the CRS reference point
- **reference_point** – CRS reference point in 2D cartesian coordinates
- **north_direction** – north direction a 2D vertex, default is (0, 1)
- **crs** – Coordinate Reference System definition XML string, default is the definition string for EPSG:3395 World Mercator

### ImageDef

**IMAGEDEF** entity defines an image file, which can be placed by the *Image* entity.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFOBJECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'IMAGEDEF'</td>
</tr>
<tr>
<td>Factory function (1)</td>
<td>ezdxf.document.Drawing.add_image_def()</td>
</tr>
<tr>
<td>Factory function (2)</td>
<td>ezdxf.sections.objects.ObjectsSection.add_image_def()</td>
</tr>
</tbody>
</table>
Warning: Do not instantiate object classes by yourself - always use the provided factory functions!

class ezdxf.entities.ImageDef

    dxf.class_version
        Current version is 0.
    dxf.filename
        Relative (to the DXF file) or absolute path to the image file as string.
    dxf.image_size
        Image size in pixel as (x, y) tuple.
    dxf.pixel_size
        Default size of one pixel in drawing units as (x, y) tuple.
    dxf.loaded
        0 = unloaded; 1 = loaded, default = 1
    dxf.resolution_units

        +---+-------+
        | 0  | No units |
        +---+-------+
        | 2   | Centimeters |
        +---+-------+
        | 5   | Inch |
        +---+-------+

        Default = 0

ImageDefReactor

    class ezdxf.entities.ImageDefReactor

    dxf.class_version
    dxf.image_handle

DXFLayout

LAYOUT entity is part of a modelspace or paperspace layout definitions.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.PlotSettings</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'LAYOUT'</td>
</tr>
<tr>
<td>Factory function</td>
<td>internal data structure, use Layouts to manage layout objects.</td>
</tr>
</tbody>
</table>

class ezdxf.entities.DXFLayout

    dxf.name
        Layout name as shown in tabs by CAD applications

TODO
Placeholder

The `ACDBPLACEHOLDER` object for internal usage.

| Subclass of | ezdxf.entities.DXFOBJECT |
| DXF type     | 'ACDBPLACEHOLDER' |
| Factory function | ezdxf.sections.objects.ObjectsSection.add_placeholder() |

**Warning:** Do not instantiate object classes by yourself - always use the provided factory functions!

```python
class ezdxf.entities.Placeholder
```

PlotSettings

All `PLOTSETTINGS` attributes are part of the `DXFLayout` entity. I don’t know if this entity also appears as standalone entity.

| Subclass of | ezdxf.entities.DXFOBJECT |
| DXF type     | 'PLOTSETTINGS' |
| Factory function | internal data structure |

```python
class ezdxf.entities.PlotSettings
```

- `dxf.page_setup_name`
  - Page setup name

**TODO**

Sun

`SUN` entity defines properties of the sun.

| Subclass of | ezdxf.entities.DXFOBJECT |
| DXF type     | 'SUN' |
| Factory function | creating a new SUN entity is not supported |

```python
class ezdxf.entities.Sun
```

- `dxf.version`
  - Current version is 1.
- `dxf.status`
  - on = 1 or off = 0
- `dxf.color`
  - *AutoCAD Color Index (ACI)* value of the sun.
- `dxf.true_color`
  - *true color* value of the sun.
dxf.intensity
   Intensity value in the range of 0 to 1. (float)

dxf.julian_day
   use calendardate() to convert dxf.julian_day to datetime.datetime object.

dxf.time
   Day time in seconds past midnight. (int)

dxf.daylight_savings_time

dxf.shadows

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sun do not cast shadows</td>
</tr>
<tr>
<td>1</td>
<td>Sun do cast shadows</td>
</tr>
</tbody>
</table>

dxf.shadow_type

dxf.shadow_map_size

dxf.shadow_softness

UnderlayDefinition

UnderlayDefinition (DXF Reference) defines an underlay file, which can be placed by the Underlay entity.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>internal base class</td>
</tr>
<tr>
<td>Factory function (1)</td>
<td>ezdxf.document.Drawing.add_underlay_def()</td>
</tr>
<tr>
<td>Factory function (2)</td>
<td>ezdxf.sections.objects.ObjectsSection.add_underlay_def()</td>
</tr>
</tbody>
</table>

class ezdxf.entities.UnderlayDefinition
   Base class of PdfDefinition, DwfDefinition and DgnDefinition

   dxf.filename
      Relative (to the DXF file) or absolute path to the underlay file as string.

   dxf.name
      Defines which part of the underlay file to display.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>'pdf'</td>
<td>PDF page number</td>
</tr>
<tr>
<td>'dgn'</td>
<td>always 'default'</td>
</tr>
<tr>
<td>'dwf'</td>
<td>?</td>
</tr>
</tbody>
</table>

Warning: Do not instantiate object classes by yourself - always use the provided factory functions!
PdfDefinition

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.UnderlayDefinition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'PDFDEFINITION'</td>
</tr>
<tr>
<td>Factory function (1)</td>
<td>ezdxf.document.Drawing.add_underlay_def()</td>
</tr>
<tr>
<td>Factory function (2)</td>
<td>ezdxf.sections.objects.ObjectsSection.add_underlay_def()</td>
</tr>
</tbody>
</table>

class ezdxf.entities.PdfDefinition
PDF underlay file.

DwfDefinition

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.UnderlayDefinition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'DWFDEFINITION'</td>
</tr>
<tr>
<td>Factory function (1)</td>
<td>ezdxf.document.Drawing.add_underlay_def()</td>
</tr>
<tr>
<td>Factory function (2)</td>
<td>ezdxf.sections.objects.ObjectsSection.add_underlay_def()</td>
</tr>
</tbody>
</table>

class ezdxf.entities.DwfDefinition
DWF underlay file.

DgnDefinition

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.UnderlayDefinition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'DGNDEFINITION'</td>
</tr>
<tr>
<td>Factory function (1)</td>
<td>ezdxf.document.Drawing.add_underlay_def()</td>
</tr>
<tr>
<td>Factory function (2)</td>
<td>ezdxf.sections.objects.ObjectsSection.add_underlay_def()</td>
</tr>
</tbody>
</table>

class ezdxf.entities.DgnDefinition
DGN underlay file.

XRecord

Important class for storing application defined data in DXF files.

XRECORD objects are used to store and manage arbitrary data. They are composed of DXF group codes ranging from 1 through 369. This object is similar in concept to XDATA but is not limited by size or order.

To reference a XRECORD by an DXF entity, store the handle of the XRECORD in the XDATA section, application defined data or the ExtensionDict of the DXF entity.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'XRECORD'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.sections.objects.ObjectsSection.add_xrecord()</td>
</tr>
</tbody>
</table>

Warning: Do not instantiate object classes by yourself - always use the provided factory functions!
class ezdxf.entities.XRecord

dxf.cloning

Duplicate record cloning flag (determines how to merge duplicate entries, ignored by ezdxf):

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>not applicable</td>
</tr>
<tr>
<td>1</td>
<td>keep existing</td>
</tr>
<tr>
<td>2</td>
<td>use clone</td>
</tr>
<tr>
<td>3</td>
<td>&lt;xref&gt;$0$&lt;name&gt;</td>
</tr>
<tr>
<td>4</td>
<td>$0$&lt;name&gt;</td>
</tr>
<tr>
<td>5</td>
<td>Unmangle name</td>
</tr>
</tbody>
</table>

tags

Raw DXF tag container Tags. Be careful ezdxf does not validate the content of XRECORDS.

### 6.8.3 Data Query

See also:

For usage of the query features see the tutorial: Tutorial for getting data from DXF files

#### Entity Query String

```
QueryString := EntityQuery ("[" AttribQuery "]" "i"?)*
```

The query string is the combination of two queries, first the required entity query and second the optional attribute query, enclosed in square brackets, append 'i.' after the closing square bracket to ignore case for strings.

#### Entity Query

The entity query is a whitespace separated list of DXF entity names or the special name '*'. Where '*' means all DXF entities, exclude some entity types by appending their names with a preceding '!' (e.g. all entities except LINE = '* !LINE'). All DXF names have to be uppercase.

#### Attribute Query

The optional attribute query is a boolean expression, supported operators are:

- not (!): !term is true, if term is false
- and (&): term & term is true, if both terms are true
- or (|): term | term is true, if one term is true
- and arbitrary nested round brackets
- append (i) after the closing square bracket to ignore case for strings

Attribute selection is a term: “name comparator value”, where name is a DXF entity attribute in lowercase, value is a integer, float or double quoted string, valid comparators are:

- "==" equal “value”
- "!=" not equal “value”
• "<" lower than "value"
• "<=" lower or equal than "value"
• ">" greater than "value"
• ">=" greater or equal than "value"
• "?" match regular expression "value"
• "!?" does not match regular expression "value"

**Query Result**

The `EntityQuery` class is the return type of all `query()` methods. `EntityQuery` contains all DXF entities of the source collection, which matches one name of the entity query AND the whole attribute query. If a DXF entity does not have or support a required attribute, the corresponding attribute search term is `False`.

**examples:**

- `LINE[text ? ".*"]`: always empty, because the LINE entity has no text attribute.
- `LINE CIRCLE[layer=="construction"]`: all LINE and CIRCLE entities with layer == "construction"
- `*[!(layer=="construction" & color<7)]`: all entities except those with layer == "construction" and color < 7
- `*[layer=="construction"]i, (ignore case) all entities with layer == "construction" | "Construction" | "ConStruction" ...

**EntityQuery Class**

```
class ezdxf.query.EntityQuery
    The `EntityQuery` class is a result container, which is filled with dxf entities matching the query string. It is possible to add entities to the container (extend), remove entities from the container and to filter the container. Supports the standard Python Sequence methods and protocols.

    first
        First entity or None.

    last
        Last entity or None.

    __len__ () → int
        Returns count of DXF entities.

    __getitem__ (item: int) → DXFEntity
        Returns DXFEntity at index `item`, supports negative indices and slicing.

    __iter__ () → Iterable[ezdxf.entities.dxfentity.DXFEntity]
        Returns iterable of DXFEntity objects.

    extend(entities: Iterable[DXFEntity], query: str = '*', unique: bool = True) → ezdxf.query.EntityQuery
        Extent the `EntityQuery` container by entities matching an additional query.

    remove(query: str = '*') → None
        Remove all entities from `EntityQuery` container matching this additional query.
```
query \( query: \text{str} = '*') \rightarrow \text{ezdxf.query.EntityQuery}

Returns a new \text{EntityQuery} container with all entities matching this additional query.

raises: ParseException (pyparsing.py)

\text{groupby} \( \text{dxfattrib: str} = '', \text{key: Callable[[DXFEntity], Hashable]} = \text{None}) \rightarrow \text{Dict[Hashable, List[DXFEntity]]}

Returns a dict of entity lists, where entities are grouped by a DXF attribute or a key function.

Parameters

- \text{dxfattrib} – grouping DXF attribute as string like 'layer'
- \text{key} – key function, which accepts a DXFEntity as argument, returns grouping key of this entity or \text{None} for ignore this object. Reason for ignoring: a queried DXF attribute is not supported by this entity

The new() Function

\text{ezdxf.query.new} \( \text{entities: Iterable['DXFEntity']} = \text{None}, \text{query: str} = '*') \rightarrow \text{EntityQuery}

Start a new query based on sequence \text{entities}. The \text{entities} argument has to be an iterable of \text{DXFEntity} or inherited objects and returns an \text{EntityQuery} object.

See also:

For usage of the groupby features see the tutorial: \text{Retrieve entities by groupby() function}

Groupby Function

\text{ezdxf.groupby.groupby} \( \text{entities: Iterable[DXFEntity]}, \text{dxfattrib: str} = '', \text{key: KeyFunc} = \text{None}) \rightarrow \text{Dict[Hashable, List[DXFEntity]]}

Groups a sequence of DXF entities by a DXF attribute like 'layer', returns a dict with \text{dxfattrib} values as key and a list of entities matching this \text{dxfattrib}. A \text{key} function can be used to combine some DXF attributes (e.g. layer and color) and should return a hashable data type like a tuple of strings, integers or floats, \text{key} function example:

```python
def group_key(entity: DXFEntity):
    return entity.dxf.layer, entity.dxf.color
```

For not suitable DXF entities return \text{None} to exclude this entity, in this case it's not required, because \text{groupby()} catches \text{DXFAttributeError} exceptions to exclude entities, which do not provide layer and/or color attributes, automatically.

Result dict for \text{dxfattrib} = 'layer' may look like this:

```
{  
    '0': [ ... list of entities ],  
    'ExampleLayer1': [ ... ],  
    'ExampleLayer2': [ ... ],  
    ...  
  }
```

Result dict for \text{key} = \text{group_key}, which returns a (layer, color) tuple, may look like this:

```
{  
    ('0', 1): [ ... list of entities ],  
    ('0', 3): [ ... ],  
}
```

(continues on next page)
All entity containers (modelspace, paperspace layouts and blocks) and the `EntityQuery` object have a dedicated `groupby()` method.

**Parameters**

- `entities` – sequence of DXF entities to group by a DXF attribute or a `key` function
- `dxfattrib` – grouping DXF attribute like 'layer'
- `key` – key function, which accepts a `DXFEntity` as argument and returns a hashable grouping key or `None` to ignore this entity.

### 6.8.4 Math Utilities

Utility functions and classes located in module `ezdxf.math`.

**Functions**

`ezdxf.math.is_close_points(p1: Vertex, p2: Vertex, abs_tol=1e-10) -> bool`

Returns `True` if `p1` is very close to `p2`.

**Parameters**

- `p1` – first vertex as `Vec3` compatible object
- `p2` – second vertex as `Vec3` compatible object
- `abs_tol` – absolute tolerance

**Raises** `TypeError` – for incompatible vertices

`ezdxf.math.closest_point(base: Vertex, points: Iterable[Vertex]) -> Vec3`

Returns closest point to `base`.

**Parameters**

- `base` – base point as `Vec3` compatible object
- `points` – iterable of points as `Vec3` compatible object

`ezdxf.math.uniform_knot_vector(count: int, order: int, normalize=False) -> List[float]`

Returns an uniform knot vector for a B-spline of `order` and `count` control points.

`order = degree + 1`

**Parameters**

- `count` – count of control points
- `order` – spline order
- `normalize` – normalize values in range [0, 1] if `True`
ezdxf.math.\texttt{open\_uniform\_knot\_vector}(\texttt{count: int, order: int, normalize=False}) \rightarrow \texttt{List[float]}

Returns an open (clamped) uniform knot vector for a B-spline of \texttt{order} and \texttt{count} control points.

\texttt{order} = degree + 1

**Parameters**
- \texttt{count} – count of control points
- \texttt{order} – spline order
- \texttt{normalize} – normalize values in range [0, 1] if \texttt{True}

ezdxf.math.\texttt{required\_knot\_values}(\texttt{count: int, order: int}) \rightarrow \texttt{int}

Returns the count of required knot values for a B-spline of \texttt{order} and \texttt{count} control points.

**Parameters**
- \texttt{count} – count of control points, in text-books referred as “n + 1”
- \texttt{order} – order of B-Spline, in text-books referred as “k”

Relationship:
“p” is the degree of the B-spline, text-book notation.
- \texttt{\texttt{k} = p + 1}
- \texttt{\texttt{2 k n + 1}}

ezdxf.math.\texttt{xround}(\texttt{value: float, rounding: float = 0.0}) \rightarrow \texttt{float}

Extended rounding function, argument \texttt{rounding} defines the rounding limit:

<table>
<thead>
<tr>
<th>\texttt{rounding}</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>remove fraction</td>
</tr>
<tr>
<td>0.1</td>
<td>round next to \texttt{x.1, x.2, \ldots x.0}</td>
</tr>
<tr>
<td>0.25</td>
<td>round next to \texttt{x.25, x.50, x.75 or x.00}</td>
</tr>
<tr>
<td>0.5</td>
<td>round next to \texttt{x.5 or x.0}</td>
</tr>
<tr>
<td>1.0</td>
<td>round to a multiple of 1: remove fraction</td>
</tr>
<tr>
<td>2.0</td>
<td>round to a multiple of 2: \texttt{xxx2, xxx4, xxx6}</td>
</tr>
<tr>
<td>5.0</td>
<td>round to a multiple of 5: \texttt{xxx5 or xxx0}</td>
</tr>
<tr>
<td>10.0</td>
<td>round to a multiple of 10: \texttt{xx10, xx20, \ldots}</td>
</tr>
</tbody>
</table>

**Parameters**
- \texttt{value} – float value to round
- \texttt{rounding} – rounding limit

ezdxf.math.\texttt{linspace}(\texttt{start: float, stop: float, num: int, endpoint=True}) \rightarrow \texttt{Iterable[float]}

Return evenly spaced numbers over a specified interval, like numpy.linspace().

Returns \texttt{num} evenly spaced samples, calculated over the interval [\texttt{start}, \texttt{stop}]. The endpoint of the interval can optionally be excluded.

ezdxf.math.\texttt{area}(\texttt{vertices: Iterable[Vertex]}) \rightarrow \texttt{float}

Returns the area of a polygon, returns the projected area in the xy-plane for 3D vertices.

ezdxf.math.\texttt{arc\_angle\_span\_deg}(\texttt{start: float, end: float}) \rightarrow \texttt{float}

Returns the counter clockwise angle span from \texttt{start} to \texttt{end} in degrees.

Returns the angle span in the range of [0, 360], 360 is a full circle. Full circle handling is a special case, because normalization of angles which describe a full circle would return 0 if treated as regular angles. e.g. (0, 360) \rightarrow
360, (0, -360) → 360, (180, -180) → 360. Input angles with the same value always return 0 by definition: (0, 0) → 0, (-180, -180) → 0, (360, 360) → 0.

```python
ezdxf.math.ellipse_param_span(start_param: float, end_param: float) → float
```

Returns the counter clockwise params span of an elliptic arc from start- to end param.

Returns the param span in the range \([0, 2\pi]\), \(2\pi\) is a full ellipse. Full ellipse handling is a special case, because normalization of params which describe a full ellipse would return 0 if treated as regular params. e.g. \((0, 2\pi) \rightarrow 2\pi, (0, -2\pi) \rightarrow 2\pi, (\pi, -\pi) \rightarrow 2\pi\). Input params with the same value always return 0 by definition: \((0, 0) \rightarrow 0, (-\pi, -\pi) \rightarrow 0, (2\pi, 2\pi) \rightarrow 0\).

### Bulge Related Functions

See also:

Description of the Bulge value.

```python
ezdxf.math.bulge_center(start_point: Vertex, end_point: Vertex, bulge: float) → Vec2
```

Returns center of arc described by the given bulge parameters.

Based on Bulge Center by Lee Mac.

**Parameters**

- `start_point` – start point as Vec2 compatible object
- `end_point` – end point as Vec2 compatible object
- `bulge` – bulge value as float

```python
ezdxf.math.bulge_radius(start_point: Vertex, end_point: Vertex, bulge: float) → float
```

Returns radius of arc defined by the given bulge parameters.

Based on Bulge Radius by Lee Mac

**Parameters**

- `start_point` – start point as Vec2 compatible object
- `end_point` – end point as Vec2 compatible object
- `bulge` – bulge value

```python
ezdxf.math.arc_to_bulge(center: Vertex, start_angle: float, end_angle: float, radius: float) → Tuple[Vec2, Vec2, float]
```

Returns bulge parameters from arc parameters.

**Parameters**

- `center` – circle center point as Vec2 compatible object
- `start_angle` – start angle in radians
- `end_angle` – end angle in radians
- `radius` – circle radius

**Returns** (start_point, end_point, bulge)

**Return type** tuple

```python
ezdxf.math.bulge_to_arc(start_point: Vertex, end_point: Vertex, bulge: float) → Tuple[Vec2, float, float]
```

Returns arc parameters from bulge parameters.
The arcs defined by bulge values of `LWPolyline` and 2D `Polyline` entities start at the vertex which includes the bulge value and ends at the following vertex.

Based on Bulge to Arc by Lee Mac.

**Parameters**
- **start_point** – start vertex as `Vec2` compatible object
- **end_point** – end vertex as `Vec2` compatible object
- **bulge** – bulge value

**Returns** (center, start_angle, end_angle, radius)

**Return type** Tuple

```python
ezdxf.math.bulge_3_points(start_point: Vertex, end_point: Vertex, point: Vertex) → float
```

Returns bulge value defined by three points.

Based on 3-Points to Bulge by Lee Mac.

**Parameters**
- **start_point** – start point as `Vec2` compatible object
- **end_point** – end point as `Vec2` compatible object
- **point** – arbitrary point as `Vec2` compatible object

### 2D Functions

```python
ezdxf.math.arc_segment_count(radius: float, angle: float, sagitta: float) → int
```

Returns the count of required segments for the approximation of an arc for a given maximum `sagitta`.

**Parameters**
- **radius** – arc radius
- **angle** – angle span of the arc in radians
- **sagitta** – max. distance from the center of an arc segment to the center of its chord

New in version 0.14.

```python
ezdxf.math.arc_chord_length(radius: float, sagitta: float) → float
```

Returns the chord length for an arc defined by `radius` and the `sagitta`.

**Parameters**
- **radius** – arc radius
- **sagitta** – distance from the center of the arc to the center of its base

New in version 0.14.

```python
ezdxf.math.distance_point_line_2d(point: Vec2, start: Vec2, end: Vec2) → float
```

Returns the normal distance from `point` to 2D line defined by `start`- and `end` point.

```python
ezdxf.math.point_to_line_relation(point: Vec2, start: Vec2, end: Vec2, abs_tol=1e-10) → int
```

Returns -1 if `point` is left line, +1 if `point` is right of `line` and 0 if `point` is on the `line`. The `line` is defined by two vertices given as arguments `start` and `end`.

**Parameters**
- **point** – 2D point to test as `Vec2`
• **start** – line definition point as `Vec2`
• **end** – line definition point as `Vec2`
• **abs_tol** – tolerance for minimum distance to line

```python
ezdxf.math.is_point_on_line_2d(point: Vec2, start: Vec2, end: Vec2, ray=True, abs_tol=1e-10) → bool
```

Returns `True` if `point` is on `line`.

**Parameters**

- **point** – 2D point to test as `Vec2`
- **start** – line definition point as `Vec2`
- **end** – line definition point as `Vec2`
- **ray** – if `True` point has to be on the infinite ray, if `False` point has to be on the line segment
- **abs_tol** – tolerance for on line test

```python
ezdxf.math.is_point_left_of_line(point: Vec2, start: Vec2, end: Vec2, colinear=False) → bool
```

Returns `True` if `point` is “left of line” defined by `start`- and `end` point, a colinear point is also “left of line” if argument `colinear` is `True`.

**Parameters**

- **point** – 2D point to test as `Vec2`
- **start** – line definition point as `Vec2`
- **end** – line definition point as `Vec2`
- **colinear** – a colinear point is also “left of line” if `True`

```python
ezdxf.math.is_point_in_polygon_2d(point: Vec2, polygon: Iterable[Vec2], abs_tol=1e-10) → int
```

Test if `point` is inside `polygon`.

**Parameters**

- **point** – 2D point to test as `Vec2`
- **polygon** – iterable of 2D points as `Vec2`
- **abs_tol** – tolerance for distance check

Returns `+1` for inside, `0` for on boundary line, `-1` for outside

```python
ezdxf.math.convex_hull_2d(points: Iterable[Vertex]) → List[Vertex]
```

Returns 2D convex hull for `points`.

**Parameters**

- **points** – iterable of points as `Vec3` compatible objects, z-axis is ignored

```python
ezdxf.math.intersection_line_line_2d(line1: Sequence[Vec2], line2: Sequence[Vec2], virtual=True, abs_tol=1e-10) → Optional[Vec2]
```

Compute the intersection of two lines in the xy-plane.

**Parameters**

- **line1** – start- and end point of first line to test e.g. `((x1, y1), (x2, y2))`.
- **line2** – start- and end point of second line to test e.g. `((x3, y3), (x4, y4))`.
- **virtual** – `True` returns any intersection point, `False` returns only real intersection points.
- **abs_tol** – tolerance for intersection test.
Returns None if there is no intersection point (parallel lines) or intersection point as Vec2

ezdxf.math.rytz_axis_construction(d1: Vec3, d2: Vec3) \rightarrow Tuple[Vec3, Vec3, float]
The Rytz’s axis construction is a basic method of descriptive Geometry to find the axes, the semi-major axis and semi-minor axis, starting from two conjugated half-diameters.


Given conjugated diameter \(d1\) is the vector from center C to point P and the given conjugated diameter \(d2\) is the vector from center C to point Q. Center of ellipse is always \((0, 0, 0)\). This algorithm works for 2D/3D vectors.

Parameters

- \(d1\) – conjugated semi-major axis as Vec3
- \(d2\) – conjugated semi-minor axis as Vec3

Returns Tuple of (major axis, minor axis, ratio)

ezdxf.math.clip_polygon_2d(clip: Iterable[Vertex], subject: Iterable[Vertex], ccw_check: bool = True) \rightarrow List[Vec2]
Clip the subject polygon by the convex clipping polygon clip.

Implements the Sutherland–Hodgman algorithm for clipping polygons.

Parameters

- \(clip\) – the convex clipping polygon as iterable of vertices
- \(subject\) – the polygon to clip as a iterable of vertices
- \(ccw\_check\) – check if the clipping polygon is in counter clockwise orientation if True, set to False if the ccw check is done by the caller

Returns the clipped subject as list of Vec2

New in version 0.16.

ezdxf.math.offset_vertices_2d(vertices: Iterable[Vertex], offset: float, closed: bool = False) \rightarrow Iterable[Vec2]
Yields vertices of the offset line to the shape defined by vertices. The source shape consist of straight segments and is located in the xy-plane, the z-axis of input vertices is ignored. Takes closed shapes into account if argument closed is True, which yields intersection of first and last offset segment as first vertex for a closed shape. For closed shapes the first and last vertex can be equal, else an implicit closing segment from last to first vertex is added. A shape with equal first and last vertex is not handled automatically as closed shape.

Warning: Adjacent collinear segments in opposite directions, same as a turn by 180 degree (U-turn), leads to unexpected results.

Parameters

- \(vertices\) – source shape defined by vertices
- \(offset\) – line offset perpendicular to direction of shape segments defined by vertices order, offset > 0 is ‘left’ of line segment, offset < 0 is ‘right’ of line segment
- \(closed\) – True to handle as closed shape

source = [(0, 0), (3, 0), (3, 3), (0, 3)]
result = list(offset_vertices_2d(source, offset=0.5, closed=True))
Example for a closed collinear shape, which creates 2 additional vertices and the first one has an unexpected location:

```python
source = [(0, 0), (0, 1), (0, 2), (0, 3)]
result = list(offset_vertices_2d(source, offset=0.5, closed=True))
```
3D Functions

See also:

The free online book 3D Math Primer for Graphics and Game Development is a very good resource for learning vector math and other graphic related topics, it is easy to read for beginners and especially targeted to programmers.

```python
ezdxf.math.basic_transformation(move: Vertex = (0, 0, 0), scale: Vertex = (1, 1, 1), z_rotation: float = 0) → Matrix44
```

Returns a combined transformation matrix for translation, scaling and rotation about the z-axis.

**Parameters**

- `move` – translation vector
- `scale` – x-, y- and z-axis scaling as float triplet, e.g. (2, 2, 1)
- `z_rotation` – rotation angle about the z-axis in radians

```python
ezdxf.math.normal_vector_3p(a: Vec3, b: Vec3, c: Vec3) → Vec3
```

Returns normal vector for 3 points, which is the normalized cross product for: `a->b x a->c`.

```python
ezdxf.math.best_fit_normal(vertices: Iterable[Vertex]) → Vec3
```

Returns the “best fit” normal for a plane defined by three or more vertices. This function tolerates imperfect plane vertices. Safe function to detect the extrusion vector of flat arbitrary polygons.

```python
ezdxf.math.is_planar_face(face: Sequence[Vec3], abs_tol=1e-9) → bool
```

Returns `True` if sequence of vectors is a planar face.

**Parameters**

- `face` – sequence of `Vec3` objects
- `abs_tol` – tolerance for normals check

```python
ezdxf.math.subdivide_face(face: Sequence[Union[Vec3, Vec2]], quads=True) → Iterable[List[Vec3]]
```

Yields new subdivided faces. Creates new faces from subdivided edges and the face midpoint by linear interpolation.

**Parameters**

- `face` – a sequence of vertices, `Vec2` and `Vec3` objects supported.
- `quads` – create quad faces if `True` else create triangles
ezdxf.math.subdivide_ngons(faces: Iterable[Sequence[Union[Vec3, Vec2]]]) → Iterable[List[Vec3]]
Yields only triangles or quad faces, subdivides ngons into triangles.

Parameters

- **faces**: iterable of faces as sequence of Vec2 and Vec3 objects

ezdxf.math.distance_point_line_3d(point: Vec3, start: Vec3, end: Vec3) → float
Returns the normal distance from point to 3D line defined by start- and end point.

ezdxf.math.intersection_ray_ray_3d(ray1: Tuple[Vec3, Vec3], ray2: Tuple[Vec3, Vec3], abs_tol=1e-10) → Sequence[Vec3]
Calculate intersection of two 3D rays, returns a 0-tuple for parallel rays, a 1-tuple for intersecting rays and a 2-tuple for not intersecting and not parallel rays with points of closest approach on each ray.

Parameters

- **ray1**: first ray as tuple of two points as Vec3 objects
- **ray2**: second ray as tuple of two points as Vec3 objects
- **abs_tol**: absolute tolerance for comparisons

ezdxf.math.estimate_tangents(points: List[Vec3], method: str = '5-points', normalize = True) → List[Vec3]
Estimate tangents for curve defined by given fit points. Calculated tangents are normalized (unit-vectors).

Available tangent estimation methods:

- “3-points”: 3 point interpolation
- “5-points”: 5 point interpolation
- “bezier”: tangents from an interpolated cubic bezier curve
- “diff”: finite difference

Parameters

- **points**: start-, end- and passing points of curve
- **method**: tangent estimation method
- **normalize**: normalize tangents if True

Returns
tangents as list of Vec3 objects

ezdxf.math.estimate_end_tangent_magnitude(points: List[Vec3], method: str = 'chord') → List[Vec3]
Estimate tangent magnitude of start- and end tangents.

Available estimation methods:

- “chord”: total chord length, curve approximation by straight segments
- “arc”: total arc length, curve approximation by arcs
- “bezier-n”: total length from cubic bezier curve approximation, n segments per section

Parameters

- **points**: start-, end- and passing points of curve
- **method**: tangent magnitude estimation method
Returns a cubic BSpline from fit points as close as possible to common CAD applications like BricsCAD.

There exist infinite numerical correct solution for this setup, but some facts are known:

• Global curve interpolation with start- and end derivatives, e.g. 6 fit points creates 8 control vertices in BricsCAD

• Degree of B-spline is always 3, the stored degree is ignored, this is only valid for B-splines defined by fit points

• Knot parametrization method is “chord”

• Knot distribution is “natural”

The last missing parameter is the start- and end tangents estimation method used by BricsCAD, if these tangents are stored in the DXF file provide them as argument tangents as 2-tuple (start, end) and the interpolated control vertices will match the BricsCAD calculation, except for floating point imprecision.

If the end tangents are not given, the start- and end tangent directions will be estimated. The argument estimate lets choose from different estimation methods (first 3 letters are significant):

• “3-points”: 3 point interpolation

• “5-points”: 5 point interpolation

• “bezier”: tangents from an interpolated cubic bezier curve

• “diff”: finite difference

The estimation method “5-p” yields the closest match to the BricsCAD rendering, but sometimes “bez” creates a better result.

If I figure out how BricsCAD estimates the end tangents directions, the argument estimate gets an additional value for that case. The existing estimation methods will perform the same way as now, except for bug fixes. But the default value may change, therefore set argument estimate to specific value to always get the same result in the future.

Parameters

• fit_points – points the spline is passing through

• tangents – start- and end tangent, default is autodetect

• estimate – tangent direction estimation method

Changed in version 0.16: removed unused arguments degree and method

Returns a cubic BSpline from fit points without end tangents.

This function uses the cubic Bézier interpolation to create multiple Bézier curves and combine them into a single B-spline, this works for short simple splines better than the fit_points_to_cad_cv(), but is worse for longer and more complex splines.

Parameters fit_points – points the spline is passing through

New in version 0.16.

B-spline interpolation by the Global Curve Interpolation. Given are the fit points and the degree of the B-spline.

The function provides 3 methods for generating the parameter vector t:
• “uniform”: creates a uniform t vector, from 0 to 1 evenly spaced, see uniform method
• “chord”, “distance”: creates a t vector with values proportional to the fit point distances, see chord length method
• “centripetal”, “sqrt_chord”: creates a t vector with values proportional to the fit point sqrt(distances), see centripetal method
• “arc”: creates a t vector with values proportional to the arc length between fit points.

It is possible to constraint the curve by tangents, by start- and end tangent if only two tangents are given or by one tangent for each fit point.

If tangents are given, they represent 1st derivatives and and should be scaled if they are unit vectors, if only start- and end tangents given the function estimate_end_tangent_magnitude() helps with an educated guess, if all tangents are given, scaling by chord length is a reasonable choice (Piegl & Tiller).

Parameters

• fit_points – fit points of B-spline, as list of Vec3 compatible objects
• tangents – if only two vectors are given, take the first and the last vector as start- and end tangent constraints or if for all fit points a tangent is given use all tangents as interpolation constraints (optional)
• degree – degree of B-spline
• method – calculation method for parameter vector t

Returns BSpline

ezdxmath.local_cubic_bspline_interpolation(fit_points: Iterable[Vertex], method: str = '5-points', tangents :Iterable[Vertex] = None) \rightarrow BSpline

B-spline interpolation by ‘Local Cubic Curve Interpolation’, which creates B-spline from fit points and estimated tangent direction at start-, end- and passing points.

Source: Piegl & Tiller: “The NURBS Book” - chapter 9.3.4

Available tangent estimation methods:

• “3-points”: 3 point interpolation
• “5-points”: 5 point interpolation
• “bezier”: cubic bezier curve interpolation
• “diff”: finite difference

or pass pre-calculated tangents, which overrides tangent estimation.

Parameters

• fit_points – all B-spline fit points as Vec3 compatible objects
• method – tangent estimation method
• tangents – tangents as Vec3 compatible objects (optional)

Returns BSpline

ezdxmath.rational_bspline_from_arc(center: Vec3 = (0, 0), radius: float=1, start_angle: float = 0, end_angle: float = 360, segments: int = 1) \rightarrow BSpline

Returns a rational B-splines for a circular 2D arc.

Parameters
• **center** – circle center as \texttt{Vec3} compatible object
• **radius** – circle radius
• **start\_angle** – start angle in degrees
• **end\_angle** – end angle in degrees
• **segments** – count of spline segments, at least one segment for each quarter (90 deg), default is 1, for as few as needed.

\texttt{ezdxf.math.rational\_bspline\_from\_ellipse(ellipse: ConstructionEllipse, segments: int = 1) \rightarrow BSpline}

Returns a rational B-splines for an elliptic arc.

**Parameters**

• **ellipse** – ellipse parameters as \texttt{ConstructionEllipse} object
• **segments** – count of spline segments, at least one segment for each quarter ($\pi/2$), default is 1, for as few as needed.

\texttt{ezdxf.math.open\_uniform\_bspline(control\_points: Iterable[Vertex], order: int = 4, weights: Iterable[float] = None) \rightarrow BSpline}

Creates an open uniform (periodic) B-spline curve (open curve).

This is an unclamped curve, which means the curve passes none of the control points.

**Parameters**

• **control\_points** – iterable of control points as \texttt{Vec3} compatible objects
• **order** – spline order (degree + 1)
• **weights** – iterable of weight values

\texttt{ezdxf.math.closed\_uniform\_bspline(control\_points: Iterable[Vertex], order: int = 4, weights: Iterable[float] = None) \rightarrow BSpline}

Creates an closed uniform (periodic) B-spline curve (open curve).

This B-spline does not pass any of the control points.

**Parameters**

• **control\_points** – iterable of control points as \texttt{Vec3} compatible objects
• **order** – spline order (degree + 1)
• **weights** – iterable of weight values

\texttt{ezdxf.math.cubic\_bezier\_from\_arc(center: Vec3 = (0, 0), radius: float=1, start\_angle: float = 0, end\_angle: float = 360, segments: int = 1) \rightarrow Iterable[Bezier4P]}

Returns an approximation for a circular 2D arc by multiple cubic Bézier-curves.

**Parameters**

• **center** – circle center as \texttt{Vec3} compatible object
• **radius** – circle radius
• **start\_angle** – start angle in degrees
• **end\_angle** – end angle in degrees
• **segments** – count of Bézier-curve segments, at least one segment for each quarter (90 deg), 1 for as few as possible.
ezdxf.math.cubic_bezier_from_ellipse (ellipse: ConstructionEllipse, segments: int = 1) → Iterable[Bezier4P]

Returns an approximation for an elliptic arc by multiple cubic Bézier-curves.

Parameters

• ellipse – ellipse parameters as ConstructionEllipse object

• segments – count of Bézier-curve segments, at least one segment for each quarter (\(\pi/2\)), 1 for as few as possible.

ezdxf.math.cubic_bezier_interpolation (points: Iterable[Vertex]) → List[Bezier4P]

Returns an interpolation curve for given data points as multiple cubic Bézier-curves. Returns n-1 cubic Bézier-curves for n given data points, curve i goes from point[i] to point[i+1].

Parameters points – data points

ezdxf.math.quadratic_to_cubic_bezier (bezier: Bezier3P) → Bezier4P


ezdxf.math.bezier_to_bspline (Iterable[Union[Bezier3P, Bezier4P]], segments int = 4) → BSpline

Convert multiple quadratic or cubic Bèzier curves into a single cubic B-spline (ezdxf.math.BSpline). For good results the curves must be lined up seamlessly, i.e. the starting point of the following curve must be the same as the end point of the previous curve. G1 continuity or better at the connection points of the Bézier curves is required to get best results.

ezdxf.math.have_bezier_curves_g1_continuity (b1: AnyBezier, b2 AnyBezier, g1_tol: float = 1e-4) → bool

Return True if the given adjacent bezier curves have G1 continuity.

Transformation Classes

OCS Class

class ezdxf.math.OCS (extrusion: Vertex = Vec3(0.0, 0.0, 1.0))

Establish an OCS for a given extrusion vector.

Parameters extrusion – extrusion vector.

ux x-axis unit vector

uy y-axis unit vector

uz z-axis unit vector

from_wcs (point: Vertex) → Vertex

Returns OCS vector for WCS point.

points_from_wcs (points: Iterable[Vertex]) → Iterable[Vertex]

Returns iterable of OCS vectors from WCS points.

to_wcs (point: Vertex) → Vertex

Returns WCS vector for OCS point.

points_to_wcs (points: Iterable[Vertex]) → Iterable[Vertex]

Returns iterable of WCS vectors for OCS points.
**render_axis** *(layout: BaseLayout, length: float = 1, colors: Tuple[int, int, int] = (1, 3, 5))*

Render axis as 3D lines into a `layout`.

**UCS Class**

```python
class ezdxf.math.UCS (origin: Vertex = (0, 0, 0), ux: Vertex = None, uy: Vertex = None, uz: Vertex = None)
```

Establish an user coordinate system (**UCS**). The UCS is defined by the origin and two unit vectors for the x-, y- or z-axis, all axis in **WCS**. The missing axis is the cross product of the given axis.

If x- and y-axis are `None`: ux = (1, 0, 0), uy = (0, 1, 0), uz = (0, 0, 1).

Unit vectors don’t have to be normalized, normalization is done at initialization, this is also the reason why scaling gets lost by copying or rotating.

**Parameters**

- **origin** – defines the UCS origin in world coordinates
- **ux** – defines the UCS x-axis as vector in **WCS**
- **uy** – defines the UCS y-axis as vector in **WCS**
- **uz** – defines the UCS z-axis as vector in **WCS**

**ux**

x-axis unit vector

**uy**

y-axis unit vector

**uz**

z-axis unit vector

**is_cartesian**

Returns `True` if cartesian coordinate system.

**copy** () → UCS

Returns a copy of this UCS.

**to_wcs** *(point: ezdxf.math._vector.Vec3) → ezdxf.math._vector.Vec3*

Returns WCS point for UCS `point`.

**points_to_wcs** *(points: Iterable[Vec3]) → Iterable[ezdxf.math._vector.Vec3]*

Returns iterable of WCS vectors for UCS `points`.

**direction_to_wcs** *(vector: ezdxf.math._vector.Vec3) → ezdxf.math._vector.Vec3*

Returns WCS direction for UCS `vector` without origin adjustment.

**from_wcs** *(point: ezdxf.math._vector.Vec3) → ezdxf.math._vector.Vec3*

Returns UCS point for WCS `point`.

**points_from_wcs** *(points: Iterable[Vec3]) → Iterable[ezdxf.math._vector.Vec3]*

Returns iterable of UCS vectors from WCS `points`.

**direction_from_wcs** *(vector: ezdxf.math._vector.Vec3) → ezdxf.math._vector.Vec3*

Returns UCS vector for WCS `vector` without origin adjustment.

**to_ocs** *(point: ezdxf.math._vector.Vec3) → ezdxf.math._vector.Vec3*

Returns OCS vector for UCS `point`.

The **OCS** is defined by the z-axis of the **UCS**.
points_to_ocs(points: Iterable[Vec3]) → Iterable[ezdxf.math._vector.Vec3]
Returns iterable of OCS vectors for UCS points.

The OCS is defined by the z-axis of the UCS.

Parameters points – iterable of UCS vertices

to_ocs_angle_deg(angle: float) → float
Transforms angle from current UCS to the parent coordinate system (most likely the WCS) including the transformation to the OCS established by the extrusion vector UCS.uz.

Parameters angle – in UCS in degrees

transform(m: Matrix44) → UCS
General inplace transformation interface, returns self (floating interface).

Parameters m – 4x4 transformation matrix (ezdxf.math.Matrix44)
New in version 0.14.

rotate(axis: Vertex, angle: float) → UCS
Returns a new rotated UCS, with the same origin as the source UCS. The rotation vector is located in the origin and has WCS coordinates e.g. (0, 0, 1) is the WCS z-axis as rotation vector.

Parameters
• axis – arbitrary rotation axis as vector in WCS
• angle – rotation angle in radians

rotate_local_x(angle: float) → UCS
Returns a new rotated UCS, rotation axis is the local x-axis.

Parameters angle – rotation angle in radians

rotate_local_y(angle: float) → UCS
Returns a new rotated UCS, rotation axis is the local y-axis.

Parameters angle – rotation angle in radians

rotate_local_z(angle: float) → UCS
Returns a new rotated UCS, rotation axis is the local z-axis.

Parameters angle – rotation angle in radians

shift(delta: Vertex) → UCS
Shifts current UCS by delta vector and returns self.

Parameters delta – shifting vector

moveto(location: Vertex) → UCS
Place current UCS at new origin location and returns self.

Parameters location – new origin in WCS

static from_x_axis_and_point_in_xy(origin: Vertex, axis: Vertex, point: Vertex) → UCS
Returns an new UCS defined by the origin, the x-axis vector and an arbitrary point in the xy-plane.

Parameters
• origin – UCS origin as (x, y, z) tuple in WCS
• axis – x-axis vector as (x, y, z) tuple in WCS
• point – arbitrary point unlike the origin in the xy-plane as (x, y, z) tuple in WCS
static from_x_axis_and_point_in_xz (origin: Vertex, axis: Vertex, point: Vertex) \rightarrow UCS

Returns an new UCS defined by the origin, the x-axis vector and an arbitrary point in the xz-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in WCS
- **axis** – x-axis vector as (x, y, z) tuple in WCS
- **point** – arbitrary point unlike the origin in the xz-plane as (x, y, z) tuple in WCS

static from_y_axis_and_point_in_xy (origin: Vertex, axis: Vertex, point: Vertex) \rightarrow UCS

Returns an new UCS defined by the origin, the y-axis vector and an arbitrary point in the xy-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in WCS
- **axis** – y-axis vector as (x, y, z) tuple in WCS
- **point** – arbitrary point unlike the origin in the xy-plane as (x, y, z) tuple in WCS

static from_y_axis_and_point_in_yz (origin: Vertex, axis: Vertex, point: Vertex) \rightarrow UCS

Returns an new UCS defined by the origin, the y-axis vector and an arbitrary point in the yz-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in WCS
- **axis** – y-axis vector as (x, y, z) tuple in WCS
- **point** – arbitrary point unlike the origin in the yz-plane as (x, y, z) tuple in WCS

static from_z_axis_and_point_in_xz (origin: Vertex, axis: Vertex, point: Vertex) \rightarrow UCS

Returns an new UCS defined by the origin, the z-axis vector and an arbitrary point in the xz-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in WCS
- **axis** – z-axis vector as (x, y, z) tuple in WCS
- **point** – arbitrary point unlike the origin in the xz-plane as (x, y, z) tuple in WCS

static from_z_axis_and_point_in_yz (origin: Vertex, axis: Vertex, point: Vertex) \rightarrow UCS

Returns an new UCS defined by the origin, the z-axis vector and an arbitrary point in the yz-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in WCS
- **axis** – z-axis vector as (x, y, z) tuple in WCS
- **point** – arbitrary point unlike the origin in the yz-plane as (x, y, z) tuple in WCS

render_axis (layout: BaseLayout, length: float = 1, colors: Tuple[int, int, int] = (1, 3, 5))

Render axis as 3D lines into a layout.

Matrix44

class e兹dфх.math.Matrix44 (*args)

This is a pure Python implementation for 4x4 transformation matrices, to avoid dependency to big numerical packages like numpy, before binary wheels, installation of these packages wasn’t always easy on Windows.
The utility functions for constructing transformations and transforming vectors and points assumes that vectors are stored as row vectors, meaning when multiplied, transformations are applied left to right (e.g. vAB transforms v by A then by B).

Matrix44 initialization:

- `Matrix44()` returns the identity matrix.
- `Matrix44(values)` values is an iterable with the 16 components of the matrix.
- `Matrix44(row1, row2, row3, row4)` four rows, each row with four values.

```python
__repr__() → str
Returns the representation string of the matrix: Matrix44((col0, col1, col2, col3), (.. .), (....), (....))
```

```python
get_row(row: int) → Tuple[float, ...]
Get row as list of of four float values.

Parameters
row – row index [0 .. 3]
```

```python
set_row(row: int, values: Sequence[float]) → None
Sets the values in a row.

Parameters
• row – row index [0 .. 3]
• values – iterable of four row values
```

```python
get_col(col: int) → Tuple[float, ...]
Returns a column as a tuple of four floats.

Parameters
col – column index [0 .. 3]
```

```python
set_col(col: int, values: Sequence[float])
Sets the values in a column.

Parameters
• col – column index [0 .. 3]
• values – iterable of four column values
```

```python
copy() → Matrix44
Returns a copy of same type.
```

```python
__copy__() → Matrix44
Returns a copy of same type.
```

```python
classmethod scale(sx: float, sy: float = None, sz: float = None) → Matrix44
Returns a scaling transformation matrix. If sy is None, sy = sx, and if sz is None sz = sx.
```

```python
classmethod translate(dx: float, dy: float, dz: float) → Matrix44
Returns a translation matrix for translation vector (dx, dy, dz).
```

```python
classmethod x_rotate(angle: float) → Matrix44
Returns a rotation matrix about the x-axis.

Parameters
angle – rotation angle in radians
```

```python
classmethod y_rotate(angle: float) → Matrix44
Returns a rotation matrix about the y-axis.

Parameters
angle – rotation angle in radians
```
classmethod \texttt{z\_rotate} \((\text{angle: float}) \rightarrow \text{Matrix44}\)

Returns a rotation matrix about the z-axis.

\textbf{Parameters} \texttt{angle} – rotation angle in radians

classmethod \texttt{axis\_rotate} \((\text{axis: Vertex, angle: float}) \rightarrow \text{Matrix44}\)

Returns a rotation matrix about an arbitrary axis.

\textbf{Parameters}

- \texttt{axis} – rotation axis as \((x, y, z)\) tuple or \texttt{Vec3} object
- \texttt{angle} – rotation angle in radians

classmethod \texttt{xyz\_rotate} \((\text{angle\_x: float, angle\_y: float, angle\_z: float}) \rightarrow \text{Matrix44}\)

Returns a rotation matrix for rotation about each axis.

\textbf{Parameters}

- \texttt{angle\_x} – rotation angle about x-axis in radians
- \texttt{angle\_y} – rotation angle about y-axis in radians
- \texttt{angle\_z} – rotation angle about z-axis in radians

classmethod \texttt{shear\_xy} \((\text{angle\_x: float, angle\_y: float}) \rightarrow \text{Matrix44}\)

Returns a translation matrix for shear mapping (visually similar to slanting) in the xy-plane.

\textbf{Parameters}

- \texttt{angle\_x} – slanting angle in x direction in radians
- \texttt{angle\_y} – slanting angle in y direction in radians

classmethod \texttt{perspective\_projection} \((\text{left: float, right: float, top: float, bottom: float, near: float, far: float}) \rightarrow \text{Matrix44}\)

Returns a matrix for a 2D projection.

\textbf{Parameters}

- \texttt{left} – Coordinate of left of screen
- \texttt{right} – Coordinate of right of screen
- \texttt{top} – Coordinate of the top of the screen
- \texttt{bottom} – Coordinate of the bottom of the screen
- \texttt{near} – Coordinate of the near clipping plane
- \texttt{far} – Coordinate of the far clipping plane

classmethod \texttt{perspective\_projection\_fov} \((\text{fov: float, aspect: float, near: float, far: float}) \rightarrow \text{Matrix44}\)

Returns a matrix for a 2D projection.

\textbf{Parameters}

- \texttt{fov} – The field of view (in radians)
- \texttt{aspect} – The aspect ratio of the screen (width / height)
- \texttt{near} – Coordinate of the near clipping plane
- \texttt{far} – Coordinate of the far clipping plane

\textbf{static} \texttt{chain} \((\ast\texttt{matrices: Iterable[Matrix44]}) \rightarrow \text{Matrix44}\)

Compose a transformation matrix from one or more matrices.
static ucs (ux: Vertex, uy: Vertex, uz: Vertex) → Matrix44
Returns a matrix for coordinate transformation from WCS to UCS. For transformation from UCS to WCS, transpose the returned matrix.

Parameters
- **ux** – x-axis for UCS as unit vector
- **uy** – y-axis for UCS as unit vector
- **uz** – z-axis for UCS as unit vector
- **origin** – UCS origin as location vector

__hash__()  
Return hash(self).

__getitem__(index: Tuple[int, int])  
Get (row, column) element.

__setitem__(index: Tuple[int, int], value: float)  
Set (row, column) element.

__iter__() → Iterable[float]  
Iterates over all matrix values.

rows () → Iterable[Tuple[float, ...]]  
Iterate over rows as 4-tuples.

columns () → Iterable[Tuple[float, ...]]  
Iterate over columns as 4-tuples.

__mul__(other: Matrix44) → Matrix44  
Returns a new matrix as result of the matrix multiplication with another matrix.

__imul__(other: Matrix44) → Matrix44  
Inplace multiplication with another matrix.

transform (vector: Vertex) → ezdxf.math._vector.Vec3  
Returns a transformed vertex.

transform_direction (vector: Vertex, normalize=False) → ezdxf.math._vector.Vec3  
Returns a transformed direction vector without translation.

transform_vertices (vectors: Iterable[Vertex]) → Iterable[ezdxf.math._vector.Vec3]  
Returns an iterable of transformed vertices.

transform_directions (vectors: Iterable[Vertex], normalize=False) → Iterable[ezdxf.math._vector.Vec3]  
Returns an iterable of transformed direction vectors without translation.

transpose () → None  
Swaps the rows for columns inplace.

determinant () → float  
Returns determinant.

inverse () → None  
Calculates the inverse of the matrix.

Raises ZeroDivisionError – if matrix has no inverse.
Construction Tools

Vec3

class ezdxf.mathVec3(*args)
   This is an immutable universal 3D vector object. This class is optimized for universality not for speed. Im-
mutable means you can’t change (x, y, z) components after initialization:

   v1 = Vec3(1, 2, 3)
   v2 = v1
   v2.z = 7  # this is not possible, raises AttributeError
   v2 = Vec3(v2.x, v2.y, 7)  # this creates a new Vec3() object
   assert v1.z == 3  # and v1 remains unchanged

Vec3 initialization:
   • Vec3(), returns Vec3(0, 0, 0)
   • Vec3((x, y)), returns Vec3(x, y, 0)
   • Vec3((x, y, z)), returns Vec3(x, y, z)
   • Vec3(x, y), returns Vec3(x, y, 0)
   • Vec3(x, y, z), returns Vec3(x, y, z)

Addition, subtraction, scalar multiplication and scalar division left and right handed are supported:

   v = Vec3(1, 2, 3)
   v + (1, 2, 3) == Vec3(2, 4, 6)
   (1, 2, 3) + v == Vec3(2, 4, 6)
   v - (1, 2, 3) == Vec3(0, 0, 0)
   (1, 2, 3) - v == Vec3(0, 0, 0)
   v * 3 == Vec3(3, 6, 9)
   3 * v == Vec3(3, 6, 9)
   Vec3(3, 6, 9) / 3 == Vec3(1, 2, 3)
   -Vec3(1, 2, 3) == (-1, -2, -3)

Comparison between vectors and vectors or tuples is supported:

   Vec3(1, 2, 3) < Vec3(2, 2, 2)
   (1, 2, 3) < tuple(Vec3(2, 2, 2))  # conversion necessary
   Vec3(1, 2, 3) == (1, 2, 3)

   bool(Vec3(1, 2, 3)) is True
   bool(Vec3(0, 0, 0)) is False

x
   x-axis value

y
   y-axis value

z
   z-axis value

xy
   Vec3 as (x, y, 0), projected on the xy-plane.

xyz
   Vec3 as (x, y, z) tuple.
vec2
  Real 2D vector as Vec2 object.

magnitude
  Length of vector.

magnitude_xy
  Length of vector in the xy-plane.

magnitude_square
  Square length of vector.

is_null
  True for Vec3(0, 0, 0).

angle
  Angle between vector and x-axis in the xy-plane in radians.

angle_deg
  Returns angle of vector and x-axis in the xy-plane in degrees.

spatial_angle
  Spatial angle between vector and x-axis in radians.

spatial_angle_deg
  Spatial angle between vector and x-axis in degrees.

__str__() → str
  Return 'x, y, z' as string.

__repr__() → str
  Return 'Vec3(x, y, z)' as string.

__len__() → int
  Returns always 3.

__hash__() → int
  Returns hash value of vector, enables the usage of vector as key in set and dict.

copy() → Vec3
  Returns a copy of vector as Vec3 object.

__copy__() → Vec3
  Returns a copy of vector as Vec3 object.

__deepcopy__(memodict: dict) → Vec3
  copy.deepcopy() support.

__getitem__(index: int) → float
  Support for indexing:
  • v[0] is v.x
  • v[1] is v.y
  • v[2] is v.z

__iter__() → Iterable[float]
  Returns iterable of x-, y- and z-axis.

__abs__() → float
  Returns length (magnitude) of vector.

replace(x: float = None, y: float = None, z: float = None) → Vec3
  Returns a copy of vector with replaced x-, y- and/or z-axis.
classmethod `generate` *(items: Iterable[Vertex]) → Iterable[Vec3]*
Returns an iterable of `Vec3` objects.

classmethod `list` *(items: Iterable[Vertex]) → List[Vec3]*
Returns a list of `Vec3` objects.

classmethod `tuple` *(items: Iterable[Vertex]) → Sequence[Vec3]*
Returns a tuple of `Vec3` objects.

classmethod `from_angle` *(angle: float, length: float = 1.) → Vec3*
Returns a `Vec3` object from `angle` in radians in the xy-plane, z-axis = 0.

classmethod `from_deg_angle` *(angle: float, length: float = 1.) → Vec3*
Returns a `Vec3` object from `angle` in degrees in the xy-plane, z-axis = 0.

orthogonal *(ccw: bool = True) → Vec3*
Returns orthogonal 2D vector, z-axis is unchanged.

Parameters

- `ccw` – counter clockwise if `True` else clockwise

`lerp` *(other: Vertex, factor=.5) → Vec3*
Returns linear interpolation between `self` and `other`.

Parameters

- `other` – end point as `Vec3` compatible object
- `factor` – interpolation factor (0 = `self`, 1 = `other`, 0.5 = mid point)

`is_parallel` *(other: Vec3, abs_tolr=1e-12) → bool*
Returns `True` if `self` and `other` are parallel to vectors.

project *(other: Vertex) → Vec3*
Returns projected vector of `other` onto `self`.

normalize *(length: float = 1.) → Vec3*
Returns normalized vector, optional scaled by `length`.

reversed () → Vec3
Returns negated vector (-`self`).

isclose *(other: Vertex, abs_tol: float = 1e-12) → bool*
Returns `True` if `self` is close to `other`. Uses `math.isclose()` to compare all axis.

__neg__ () → Vec3
Returns negated vector (-`self`).

__bool__ () → bool
Returns `True` if vector is not (0, 0, 0).

__eq__ *(other: Vertex) → bool*
Equal operator.

Parameters

- `other` – `Vec3` compatible object

__lt__ *(other: Vertex) → bool*
Lower than operator.

Parameters

- `other` – `Vec3` compatible object

__add__ *(other: Vertex) → Vec3*
Add `Vec3` operator: `self + other`.

__radd__ *(other: Vertex) → Vec3*
RAdd `Vec3` operator: `other + self`.
```python
__sub__(other: Vertex) → Vec3
  Sub  Vec3 operator: self - other.
__rsub__(other: Vertex) → Vec3
  RSub  Vec3 operator: other - self.
__mul__(other: float) → Vec3
  Scalar Mul operator: self * other.
__rmul__(other: float) → Vec3
  Scalar RMul operator: other * self.
__truediv__(other: float) → Vec3
  Scalar Div operator: self / other.
dot(other: Vertex) → float
  Dot operator: self . other
  Parameters other – Vec3 compatible object
cross(other: Vertex) → Vec3
  Dot operator: self x other
  Parameters other – Vec3 compatible object
distance(other: Vertex) → float
  Returns distance between self and other vector.
angle_about(base: Vec3, target: Vec3) → float
  Returns counter clockwise angle in radians about self from base to target when projected onto the plane defined by self as the normal vector.
  Parameters
    • base – base vector, defines angle 0
    • target – target vector
angle_between(other: Vertex) → float
  Returns angle between self and other in radians. +angle is counter clockwise orientation.
  Parameters other – Vec3 compatible object
rotate(angle: float) → Vec3
  Returns vector rotated about angle around the z-axis.
  Parameters angle – angle in radians
rotate_deg(angle: float) → Vec3
  Returns vector rotated about angle around the z-axis.
  Parameters angle – angle in degrees
static sum(items: Iterable[Vertex]) → Vec3
  Add all vectors in items.
```

```python
ezdxf.math.X_AXIS
  Vec3(1, 0, 0)
ezdxf.math.Y_AXIS
  Vec3(0, 1, 0)
ezdxf.math.Z_AXIS
  Vec3(0, 0, 1)
```
ezdxf Documentation, Release 0.16.2

ezdxf.math.NULLVEC
Vec3(0, 0, 0)

Vec2

class ezdxf.math(Vec2 (v: Any, y: float = None)
Vec2 represents a special 2D vector (x, y). The Vec2 class is optimized for speed and not immutable, iadd(), isub(), imul() and idiv() modifies the vector itself, the Vec3 class returns a new object.

Vec2 initialization accepts float-tuples (x, y [, z]), two floats or any object providing x and y attributes like Vec2 and Vec3 objects.

Parameters
• v – vector object with x and y attributes/properties or a sequence of float [x, y, ...]
  or x-axis as float if argument y is not None
• y – second float for Vec2 (x, y)

Vec2 implements a subset of Vec3.

Plane

class ezdxf.math.Plane (normal: Vec3, distance: float)
Represents a plane in 3D space as normal vector and the perpendicular distance from origin.

normal
Normal vector of the plane.

distance_from_origin
The (perpendicular) distance of the plane from origin (0, 0, 0).

vector
Returns the location vector.

classmethod from_3p (a: Vec3, b: Vec3, c: Vec3) → Plane
Returns a new plane from 3 points in space.

classmethod from_vector (vector) → Plane
Returns a new plane from a location vector.

copy () → Plane
Returns a copy of the plane.

signed_distance_to (v: Vec3) → float
Returns signed distance of vertex v to plane, if distance is > 0, v is in ‘front’ of plane, in direction of the normal vector, if distance is < 0, v is at the ‘back’ of the plane, in the opposite direction of the normal vector.

distance_to (v: Vec3) → float
Returns absolute (unsigned) distance of vertex v to plane.

is_coplanar_vertex (v: Vec3, abs_tol=1e-9) → bool
Returns True if vertex v is coplanar, distance from plane to vertex v is 0.

is_coplanar_plane (p: Plane, abs_tol=1e-9) → bool
Returns True if plane p is coplanar, normal vectors in same or opposite direction.
**BoundingBox**

class ezdxf.math.BoundingBox(vertices: Iterable[Vertex] = None)
3D bounding box.

Parameters vertices – iterable of (x, y, z) tuples or Vec3 objects

extmin
“lower left” corner of bounding box

extmax
“upper right” corner of bounding box

inside(vertex: Vertex) → bool
Returns True if vertex is inside this bounding box.

Vertices at the box border are inside!

any_inside(vertices: Iterable[Vertex]) → bool
Returns True if any vertex is inside this bounding box.

Vertices at the box border are inside!

all_inside(vertices: Iterable[Vertex]) → bool
Returns True if all vertices are inside this bounding box.

Vertices at the box border are inside!

intersect(other: BoundingBox) → bool
Returns True if this bounding box intersects with other.

Touching bounding boxes do not intersect!

extend(vertices: Iterable[Vertex]) → None
Extend bounds by vertices.

Parameters vertices – iterable of Vertex objects

union(other: BoundingBox) → BoundingBox
Returns a new bounding box as union of this and other bounding box.

**BoundingBox2d**

class ezdxf.math.BoundingBox2d(vertices: Iterable[Vertex] = None)
Optimized 2D bounding box.

Parameters vertices – iterable of (x, y[, z]) tuples or Vec3 objects

extmin
“lower left” corner of bounding box

extmax
“upper right” corner of bounding box

inside(vertex: Vertex) → bool
Returns True if vertex is inside this bounding box.

Vertices at the box border are inside!

any_inside(vertices: Iterable[Vertex]) → bool
Returns True if any vertex is inside this bounding box.

Vertices at the box border are inside!
all_inside (vertices: Iterable[Vertex]) → bool
    Returns True if all vertices are inside this bounding box.
    Vertices at the box border are inside!

intersect (other: BoundingBox2d) → bool
    Returns True if this bounding box intersects with other.
    Touching bounding boxes do not intersect!

extend (vertices: Iterable[Vertex]) → None
    Extend bounds by vertices.

    Parameters vertices – iterable of Vertex objects

union (other: BoundingBox2d) → BoundingBox2d
    Returns a new bounding box as union of this and other bounding box.

ConstructionRay

class ezdxf.math.ConstructionRay (p1: Vertex, p2: Vertex = None, angle: float = None)
    Infinite 2D construction ray as immutable object.

    Parameters

    • p1 – definition point 1
    • p2 – ray direction as 2nd point or None
    • angle – ray direction as angle in radians or None

    location
        Location vector as Vec2.

    direction
        Direction vector as Vec2.

    slope
        Slope of ray or None if vertical.

    angle
        Angle between x-axis and ray in radians.

    angle_deg
        Angle between x-axis and ray in degrees.

    is_vertical
        True if ray is vertical (parallel to y-axis).

    is_horizontal
        True if ray is horizontal (parallel to x-axis).

    __str__ ()
        Return str(self).

    is_parallel (self, other: ConstructionRay) → bool
        Returns True if rays are parallel.

    intersect (other: ConstructionRay) → Vec2
        Returns the intersection point as (x, y) tuple of self and other.

        Raises ParallelRaysError – if rays are parallel
**orthogonal** (location: 'Vertex') → ConstructionRay
  Returns orthogonal ray at location.

**bisectrix** (other: ConstructionRay) → ConstructionRay:
  Bisectrix between self and other.

**yof** (x: float) → float
  Returns y-value of ray for x location.
  Raises ArithmeticError – for vertical rays

**xof** (y: float) → float
  Returns x-value of ray for y location.
  Raises ArithmeticError – for horizontal rays

### ConstructionLine

**class** ezdxf.math.ConstructionLine (start: Vertex, end: Vertex)
  2D ConstructionLine is similar to ConstructionRay, but has a start- and endpoint. The direction of line goes from start- to endpoint, “left of line” is always in relation to this line direction.

  **Parameters**
  
  - **start** – start point of line as Vec2 compatible object
  - **end** – end point of line as Vec2 compatible object

**start**
  start point as Vec2

**end**
  end point as Vec2

**bounding_box**
  bounding box of line as BoundingBox2d object.

**ray**
  collinear ConstructionRay.

**is_vertical**
  True if line is vertical.

**is_horizontal**
  True if line is horizontal.

**__str__**( )
  Return str(self).

**translate** (dx: float, dy: float) → None
  Move line about dx in x-axis and about dy in y-axis.

  **Parameters**
  
  - **dx** – translation in x-axis
  - **dy** – translation in y-axis

**length**( ) → float
  Returns length of line.

**midpoint**( ) → Vec2
  Returns mid point of line.
inside_bounding_box (point: Vertex) → bool
Returns True if point is inside of line bounding box.

intersect (other: ConstructionLine, abs_tol: float=1e-10) → Optional[Vec2]
Returns the intersection point of to lines or None if they have no intersection point.

Parameters
- other – other ConstructionLine
- abs_tol – tolerance for distance check

has_intersection (other: ConstructionLine, abs_tol: float=1e-10) → bool
Returns True if has intersection with other line.

is_point_left_of_line (point: Vertex, colinear=False) → bool
Returns True if point is left of construction line in relation to the line direction from start to end.
If colinear is True, a colinear point is also left of the line.

ConstructionCircle

class ezdxf.math.ConstructionCircle (center: Vertex, radius: float = 1.0)
Circle construction tool.

Parameters
- center – center point as Vec2 compatible object
- radius – circle radius > 0

center
center point as Vec2

radius
radius as float

bounding_box
2D bounding box of circle as BoundingBox2d object.

static from_3p (p1: Vertex, p2: Vertex, p3: Vertex) → ConstructionCircle
Creates a circle from three points, all points have to be compatible to Vec2 class.

__str__ () → str
Returns string representation of circle “ConstructionCircle(center, radius)”. 

translate (dx: float, dy: float) → None
Move circle about dx in x-axis and about dy in y-axis.

Parameters
- dx – translation in x-axis
- dy – translation in y-axis

point_at (angle: float) → Vec2
Returns point on circle at angle as Vec2 object.

Parameters angle – angle in radians

inside (point: Vertex) → bool
Returns True if point is inside circle.
**tangent** (*angle: float*) → *ConstructionRay*

Returns tangent to circle at *angle* as *ConstructionRay* object.

- **Parameters**
  - *angle* – angle in radians

**intersect_ray** (*ray: ConstructionRay, abs_tol: float = 1e-10*) → *Sequence[Vec2]*

Returns intersection points of circle and *ray* as sequence of *Vec2* objects.

- **Parameters**
  - *ray* – intersection ray
  - *abs_tol* – absolute tolerance for tests (e.g. test for tangents)

- **Returns**
  - tuple of *Vec2* objects

<table>
<thead>
<tr>
<th>tuple size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no intersection</td>
</tr>
<tr>
<td>1</td>
<td>ray is a tangent to circle</td>
</tr>
<tr>
<td>2</td>
<td>ray intersects with the circle</td>
</tr>
</tbody>
</table>

**intersect_circle** (*other: ConstructionCircle, abs_tol: float = 1e-10*) → *Sequence[Vec2]*

Returns intersection points of two circles as sequence of *Vec2* objects.

- **Parameters**
  - *other* – intersection circle
  - *abs_tol* – absolute tolerance for tests

- **Returns**
  - tuple of *Vec2* objects

<table>
<thead>
<tr>
<th>tuple size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no intersection</td>
</tr>
<tr>
<td>1</td>
<td>circle touches the <em>other</em> circle at one point</td>
</tr>
<tr>
<td>2</td>
<td>circle intersects with the <em>other</em> circle</td>
</tr>
</tbody>
</table>

**ConstructionArc**

class ezdxf.math.ConstructionArc(*center: Vertex = (0, 0), radius: float = 1, start_angle: float = 0, end_angle: float = 360, is_counter_clockwise: bool = True*)

This is a helper class to create parameters for the DXF *Arc* class.

*ConstructionArc* represents a 2D arc in the xy-plane, use an *UCS* to place arc in 3D space, see method `add_to_layout()`.

- Implements the 2D transformation tools: `translate()`, `scale_uniform()` and `rotate_z()`

- **Parameters**
  - *center* – center point as *Vec2* compatible object
  - *radius* – radius
  - *start_angle* – start angle in degrees
• **end_angle** – end angle in degrees

• **is_counter_clockwise** – swaps start- and end angle if `False`

center
  center point as `Vec2`

radius
  radius as float

**start_angle**
  start angle in degrees

**end_angle**
  end angle in degrees

**angle_span**
  Returns angle span of arc from start- to end param.

**start_angle_rad**
  Returns the start angle in radians.

**end_angle_rad**
  Returns the end angle in radians.

**start_point**
  start point of arc as `Vec2`.

**end_point**
  end point of arc as `Vec2`.

**bounding_box**
  bounding box of arc as `BoundingBox2d`.

**angles** *(num: int) → Iterable[float]*)
  Returns `num` angles from start- to end angle in degrees in counter clockwise order.
  All angles are normalized in the range from [0, 360).

**vertices** *(a: Iterable[float]) → Iterable[ezdxf.math._vector.Vec2]*)
  Yields vertices on arc for angles in iterable `a` in WCS as location vectors.

  **Parameters**
  `a` – angles in the range from 0 to 360 in degrees, arc goes counter clockwise around the z-axis, WCS x-axis = 0 deg.

**tangents** *(a: Iterable[float]) → Iterable[ezdxf.math._vector.Vec2]*)
  Yields tangents on arc for angles in iterable `a` in WCS as direction vectors.

  **Parameters**
  `a` – angles in the range from 0 to 360 in degrees, arc goes counter clockwise around the z-axis, WCS x-axis = 0 deg.

**translate** *(dx: float, dy: float) → ConstructionArc*
  Move arc about `dx` in x-axis and about `dy` in y-axis, returns `self` (floating interface).

  **Parameters**
  • `dx` – translation in x-axis
  • `dy` – translation in y-axis

**scale_uniform** *(s: float) → ConstructionArc*
  Scale arc inplace uniform about `s` in x- and y-axis, returns `self` (floating interface).

**rotate_z** *(angle: float) → ConstructionArc*
  Rotate arc inplace about z-axis, returns `self` (floating interface).
Parameters `angle` – rotation angle in degrees

classmethod `from_2p_angle` (start_point: Vertex, end_point: Vertex, angle: float, ccw: bool = True) \rightarrow ConstructionArc
Create arc from two points and enclosing angle. Additional precondition: arc goes by default in counter clockwise orientation from `start_point` to `end_point`, can be changed by `ccw = False`.

Parameters
- `start_point` – start point as `Vec2` compatible object
- `end_point` – end point as `Vec2` compatible object
- `angle` – enclosing angle in degrees
- `ccw` – counter clockwise direction if `True`

classmethod `from_2p_radius` (start_point: Vertex, end_point: Vertex, radius: float, ccw: bool = True, center_is_left: bool = True) \rightarrow ConstructionArc
Create arc from two points and arc radius. Additional precondition: arc goes by default in counter clockwise orientation from `start_point` to `end_point` can be changed by `ccw = False`.

The parameter `center_is_left` defines if the center of the arc is left or right of the line from `start_point` to `end_point`. Parameter `ccw = False` swaps start- and end point, which also inverts the meaning of `center_is_left`.

Parameters
- `start_point` – start point as `Vec2` compatible object
- `end_point` – end point as `Vec2` compatible object
- `radius` – arc radius
- `ccw` – counter clockwise direction if `True`
- `center_is_left` – center point of arc is left of line from start- to end point if `True`

classmethod `from_3p` (start_point: Vertex, end_point: Vertex, def_point: Vertex, ccw: bool = True) \rightarrow ConstructionArc
Create arc from three points. Additional precondition: arc goes in counter clockwise orientation from `start_point` to `end_point`.

Parameters
- `start_point` – start point as `Vec2` compatible object
- `end_point` – end point as `Vec2` compatible object
- `def_point` – additional definition point as `Vec2` compatible object
- `ccw` – counter clockwise direction if `True`

`add_to_layout` (layout: BaseLayout, ucs: UCS = None, dxfattribs: dict = None) \rightarrow Arc
Add arc as DXF `Arc` entity to a layout.

Supports 3D arcs by using an `UCS`. An `ConstructionArc` is always defined in the xy-plane, but by using an arbitrary UCS, the arc can be placed in 3D space, automatically OCS transformation included.

Parameters
- `layout` – destination layout as `BaseLayout` object
- `ucs` – place arc in 3D space by `UCS` object
- `dxfattribs` – additional DXF attributes for the DXF `Arc` entity
ConstructionEllipse

class ezdxf.math.ConstructionEllipse(center: Vertex = Vec3(0.0, 0.0, 0.0), major_axis: Vertex = Vec3(1.0, 0.0, 0.0), extrusion: Vertex = Vec3(0.0, 0.0, 1.0), ratio: float = 1, start_param: float = 0, end_param: float = 6.283185307179586, ccw: bool = True)

This is a helper class to create parameters for 3D ellipses.

Parameters

- center – 3D center point
- major_axis – major axis as 3D vector
- extrusion – normal vector of ellipse plane
- ratio – ratio of minor axis to major axis
- start_param – start param in radians
- end_param – end param in radians
- ccw – is counter clockwise flag - swaps start- and end param if False

center
center point as Vec3

major_axis
major axis as Vec3

minor_axis
minor axis as Vec3, automatically calculated from major_axis and extrusion.

extrusion
extrusion vector (normal of ellipse plane) as Vec3

ratio
ratio of minor axis to major axis (float)

start
start param in radians (float)

drop
end param in radians (float)

start_point
Returns start point of ellipse as Vec3.

drop
Returns end point of ellipse as Vec3.

to_ocs () -> ConstructionEllipse
Returns ellipse parameters as OCS representation.
OCS elevation is stored in center.z.

params (num: int) -> Iterable[float]
Returns num params from start- to end param in counter clockwise order.
All params are normalized in the range from [0, 2π).

vertices (params: Iterable[float]) -> Iterable[ezdxf.math._vector.Vec3]
Yields vertices on ellipse for iterable params in WCS.
**Parameters** `params` – param values in the range from \([0, 2\pi]\) in radians, param goes counter clockwise around the extrusion vector, major_axis = local x-axis = 0 rad.

**flattening** `(distance: float, segments: int = 4) → Iterable[ezdxf.math._vector.Vec3]`

Adaptive recursive flattening. The argument `segments` is the minimum count of approximation segments, if the distance from the center of the approximation segment to the curve is bigger than `distance` the segment will be subdivided. Returns a closed polygon for a full ellipse: start vertex == end vertex.

**Parameters**

- `distance` – maximum distance from the projected curve point onto the segment chord.
- `segments` – minimum segment count

New in version 0.15.

**params_from_vertices** `(vertices: Iterable[Vertex]) → Iterable[float]`

Yields ellipse params for all given `vertices`.

The vertex don’t has to be exact on the ellipse curve or in the range from start- to end param or even in the ellipse plane. Param is calculated from the intersection point of the ray projected on the ellipse plane from the center of the ellipse through the vertex.

**Warning:** An input for start- and end vertex at param 0 and \(2\pi\) return unpredictable results because of floating point inaccuracy, sometimes 0 and sometimes \(2\pi\).

dxfattribs () → Dict[KT, VT]

Returns required DXF attributes to build an ELLIPSE entity.

Entity ELLIPSE has always a ratio in range from 1e-6 to 1.

**main_axis_points** () → Iterable[ezdxf.math._vector.Vec3]

Yields main axis points of ellipse in the range from start- to end param.

**classmethod from_arc** `(center: Vertex=(0, 0, 0), radius: float = 1, extrusion: Vertex=(0, 0, 1), start_angle: float = 0, end_angle: float = 360, ccw: bool = True) → ConstructionEllipse`

Returns `ConstructionEllipse` from arc or circle.

Arc and Circle parameters defined in OCS.

**Parameters**

- `center` – center in OCS
- `radius` – arc or circle radius
- `extrusion` – OCS extrusion vector
- `start_angle` – start angle in degrees
- `end_angle` – end angle in degrees
- `ccw` – arc curve goes counter clockwise from start to end if True

transform `(m: Matrix44)`

Transform ellipse in place by transformation matrix \(m\).

**swap_axis** () → None

Swap axis and adjust start- and end parameter.

**add_to_layout** `(layout: BaseLayout, dxfattribs: dict = None) → Ellipse`

Add ellipse as DXF `Ellipse` entity to a layout.
Parameters

- **layout** – destination layout as `BaseLayout` object
- **dxfattribs** – additional DXF attributes for DXF `Ellipse` entity

**ConstructionBox**

class ezdxf.math.ConstructionBox(center: `Vertex = (0, 0)`, width: `float = 1`, height: `float = 1`, angle: `float = 0`)

Helper class to create rectangles.

Parameters

- **center** – center of rectangle
- **width** – width of rectangle
- **height** – height of rectangle
- **angle** – angle of rectangle in degrees

```python
center
```
box center

```python
width
```
box width

```python
height
```
box height

```python
angle
```
rotation angle in degrees

```python
corners
```
box corners as sequence of `Vec2` objects.

```python
bounding_box
```
`BoundingBox2d`

```python
incircle_radius
```
incircle radius

```python
circumcircle_radius
```
circum circle radius

```python
__iter__() → `Iterable[Vec2]`
```
Iterable of box corners as `Vec2` objects.

```python
__getitem__(corner) → `Vec2`
```
Get corner by index `corner`, list like slicing is supported.

```python
__repr__() → `str`
```
Returns string representation of box as `ConstructionBox(center, width, height, angle)`

```python
classmethod from_points(p1: `Vertex`, p2: `Vertex`) → `ConstructionBox`
```
Creates a box from two opposite corners, box sides are parallel to x- and y-axis.

Parameters

- **p1** – first corner as `Vec2` compatible object
- **p2** – second corner as `Vec2` compatible object
translate\((dx: \text{float}, dy: \text{float}) \rightarrow \text{None}\)
Move box about \(dx\) in x-axis and about \(dy\) in y-axis.

**Parameters**
- \(dx\) – translation in x-axis
- \(dy\) – translation in y-axis

expand\((dw: \text{float}, dh: \text{float}) \rightarrow \text{None}\)
Expand box: \(dw\) expand width, \(dh\) expand height.

scale\((sw: \text{float}, sh: \text{float}) \rightarrow \text{None}\)
Scale box: \(sw\) scales width, \(sh\) scales height.

rotate\((\text{angle: float}) \rightarrow \text{None}\)
Rotate box by \(\text{angle}\) in degrees.

is_inside\((\text{point: Vertex}) \rightarrow \text{bool}\)
Returns True if \(\text{point}\) is inside of box.

is_any_corner_inside\((\text{other: ConstructionBox}) \rightarrow \text{bool}\)
Returns True if any corner of \(\text{other}\) box is inside this box.

is_overlapping\((\text{other: ConstructionBox}) \rightarrow \text{bool}\)
Returns True if this box and \(\text{other}\) box do overlap.

border_lines() \rightarrow \text{Sequence[ConstructionLine]}
Returns border lines of box as sequence of \(\text{ConstructionLine}\).

intersect\((\text{line: ConstructionLine}) \rightarrow \text{List[Vec2]}\)
Returns 0, 1 or 2 intersection points between \(\text{line}\) and box border lines.

**Parameters** \(\text{line}\) – line to intersect with border lines

**Returns**

<table>
<thead>
<tr>
<th>list size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no intersection</td>
</tr>
<tr>
<td>1</td>
<td>line touches box at one corner</td>
</tr>
<tr>
<td>2</td>
<td>line intersects with box</td>
</tr>
</tbody>
</table>

**Shape2d**

class ezdxf.math.Shape2d\((\text{vertices: Iterable[Vertex]} = \text{None})\)
2D geometry object as list of \text{Vec2} objects, vertices can be moved, rotated and scaled.

**Parameters** \text{vertices} – iterable of \text{Vec2} compatible objects.

vertices
List of \text{Vec2} objects

bounding_box
\text{BoundingBox2d}

__len__() \rightarrow \text{int}
Returns count of vertices.

__getitem__\((\text{item}) \rightarrow \text{Vec2}\)
Get vertex by index \text{item}, supports list like slicing.
append(vertex: Vertex) → None
  Append single vertex.

  Parameters vertex – vertex as Vec2 compatible object

extend(vertices: Iterable[T_co]) → None
  Append multiple vertices.

  Parameters vertices – iterable of vertices as Vec2 compatible objects

translate(vector: Vertex) → None
  Translate shape about vector.

scale(sx: float = 1.0, sy: float = 1.0) → None
  Scale shape about sx in x-axis and sy in y-axis.

scale_uniform(scale: float) → None
  Scale shape uniform about scale in x- and y-axis.

rotate(angle: float, center: Vertex = None) → None
  Rotate shape around rotation center about angle in degrees.

rotate_rad(angle: float, center: Vertex = None) → None
  Rotate shape around rotation center about angle in radians.

offset(offset: float, closed: bool = False) → ezdxf.math.shape.Shape2d
  Returns a new offset shape, for more information see also ezdxf.math.offset_vertices_2d() function.

  Parameters
    • offset – line offset perpendicular to direction of shape segments defined by vertices order, offset > 0 is ‘left’ of line segment, offset < 0 is ‘right’ of line segment
    • closed – True to handle as closed shape

convex_hull() → ezdxf.math.shape.Shape2d
  Returns convex hull as new shape.

Curves

BSpline

class ezdxf.math.BSpline(control_points: Iterable[Vertex], order: int = 4, knots: Iterable[float] = None, weights: Iterable[float] = None)
  Representation of a B-spline curve. The default configuration of the knot vector is an uniform open knot vector (“clamped”).

  Factory functions:
    • fit_points_to_cad_cv()
    • fit_points_to_cubic_bezier()
    • open_uniform_bspline()
    • closed_uniform_bspline()
    • rational_uniform_bspline_from_arc()
    • rational_uniform_bspline_from_ellipse()
    • global_bspine_interpolation()
• `local_cubic_bspline_interpolation()`

**Parameters**

- `control_points` – iterable of control points as `Vec3` compatible objects
- `order` – spline order (degree + 1)
- `knots` – iterable of knot values
- `weights` – iterable of weight values

**knots()** → Tuple[float, ...]

Returns a tuple of knot values as floats, the knot vector **always** has order + count values (n + p + 2 in text book notation).

**weights()** → Tuple[float, ...]

Returns a tuple of weights values as floats, one for each control point or an empty tuple.

**params** (segments: int) → Iterable[float]

Yield evenly spaced parameters for given segment count.

**reverse()** → BSpline

Returns a new `BSpline` object with reversed control point order.

**transform** (m: `Matrix44`) → BSpline

Returns a new `BSpline` object transformed by a `Matrix44` transformation matrix.

**approximate** (segments: int = 20) → Iterable[Vec3]

Approximates curve by vertices as `Vec3` objects, vertices count = segments + 1.

**flattening** (distance: float, segments: int = 4) → Iterable[Vec3]

Adaptive recursive flattening. The argument `segments` is the minimum count of approximation segments between two knots, if the distance from the center of the approximation segment to the curve is bigger than `distance` the segment will be subdivided.

**Parameters**

- `distance` – maximum distance from the projected curve point onto the segment chord.
- `segments` – minimum segment count between two knots

New in version 0.15.

**point** (t: float) → Vec3

Returns point for parameter `t`.

Parameters `t` – parameter in range [0, max_t]

**points** (t: float) → List[Vec3]

Yields points for parameter vector `t`.

Parameters `t` – parameters in range [0, max_t]

**derivative** (t: float, n: int=2) → List[Vec3]

Return point and derivatives up to `n <= degree` for parameter `t`.

e.g. n=1 returns point and 1st derivative.

Parameters

- `t` – parameter in range [0, max_t]
- `n` – compute all derivatives up to `n <= degree`

Returns n+1 values as `Vec3` objects
derivatives \((t: \text{Iterable[float]}, \, n: \text{int}=2) \rightarrow \text{Iterable[List[Vec3]]}\)
Yields points and derivatives up to \(n \leq \text{degree}\) for parameter vector \(t\).

  e.g. \(n=1\) returns point and 1st derivative.

Parameters

  • \(t\) - parameters in range \([0, \max_t]\)
  • \(n\) - compute all derivatives up to \(n \leq \text{degree}\)

Returns List of \(n+1\) values as \(\text{Vec3}\) objects

insert_knot \((t: \text{float}) \rightarrow \text{BSpline}\)
Insert an additional knot, without altering the shape of the curve. Returns a new \(\text{BSpline}\) object.

Parameters \(t\) - position of new knot \(0 < t < \max_t\)

knot_refinement \((u: \text{Iterable[flat]}) \rightarrow \text{BSpline}\)
Insert multiple knots, without altering the shape of the curve. Returns a new \(\text{BSpline}\) object.

Parameters \(u\) - vector of new knots \(t\) and for each \(t\): \(0 < t < \max_t\)

static from_ellipse \((\text{ellipse: ConstructionEllipse}) \rightarrow \text{BSpline}\)
Returns the ellipse as \(\text{BSpline}\) of 2nd degree with as few control points as possible.

static from_arc \((\text{arc: ConstructionArc}) \rightarrow \text{BSpline}\)
Returns the arc as \(\text{BSpline}\) of 2nd degree with as few control points as possible.

static from_fit_points \((\text{points: \text{Iterable[Vertex]}, degree:\text{int}=3, method='chord'}) \rightarrow \text{BSpline}\)
Returns \(\text{BSpline}\) defined by fit points.

static arc_approximation \((\text{arc: ConstructionArc, num:\text{int}=16}) \rightarrow \text{BSpline}\)
Returns an arc approximation as \(\text{BSpline}\) with \(\text{num}\) control points.

static ellipse_approximation \((\text{ellipse: ConstructionEllipse, num:\text{int}=16}) \rightarrow \text{BSpline}\)
Returns an ellipse approximation as \(\text{BSpline}\) with \(\text{num}\) control points.

bezier_decomposition() \rightarrow \text{Iterable[List[Vec3]]}\)
Decompose a non-rational B-spline into multiple Bézier curves.

  This is the preferred method to represent the most common non-rational B-splines of 3rd degree by cubic Bézier curves, which are often supported by render backends.

  Returns Yields control points of Bézier curves, each Bézier segment has degree+1 control points
  e.g. \(\text{B-spline}\) of 3rd degree yields cubic Bézier curves of 4 control points.

  Parameters

  • \(\text{level}\) - an educated guess, the first level of approximation segments is based on the count of control points and their distribution along the B-spline, every additional level is a subdivision of the previous level.

  E.g. a B-Spline of 8 control points has 7 segments at the first level, 14 at the 2nd level and 28 at the 3rd level, a level \(\geq 3\) is recommended.

  • \(\text{segments}\) - a given count of evenly distributed approximation segments.

  Parameters
• **level** – subdivision level of approximation segments (ignored if argument `segments` is not `None`)

• **segments** – absolute count of approximation segments

**Returns** Yields control points of cubic Bézier curves as `Bezier4P` objects

### Bezier

**class** `ezdxf.math.Bezier(defpoints: Iterable[Vertex])`

A Bézier curve is a parametric curve used in computer graphics and related fields. Bézier curves are used to model smooth curves that can be scaled indefinitely. “Paths”, as they are commonly referred to in image manipulation programs, are combinations of linked Bézier curves. Paths are not bound by the limits of rasterized images and are intuitive to modify. (Source: Wikipedia)

This is a generic implementation which works with any count of definition points greater than 2, but it is a simple and slow implementation. For more performance look at the specialized `Bezier4P` class.

Objects are immutable.

**Parameters**

- **defpoints** – iterable of definition points as `Vec3` compatible objects.

**control_points**

Control points as tuple of `Vec3` objects.

**params**(segments: int) → Iterable[float]

Yield evenly spaced parameters from 0 to 1 for given segment count.

**reverse()** → Bezier

Returns a new Bézier-curve with reversed control point order.

**transform**(m: Matrix44) → Bezier

General transformation interface, returns a new Bezier curve.

**Parameters**

- **m** – 4x4 transformation matrix (`ezdxf.math.Matrix44`)

New in version 0.14.

**approximate**(segments: int = 20) → Iterable[Vec3]

Approximates curve by vertices as `Vec3` objects, vertices count = segments + 1.

**flattening**(distance: float, segments: int=4) → Iterable[Vec3]

Adaptive recursive flattening. The argument `segments` is the minimum count of approximation segments, if the distance from the center of the approximation segment to the curve is bigger than `distance` the segment will be subdivided.

**Parameters**

- **distance** – maximum distance from the center of the curve (Cn) to the center of the linear (C1) curve between two approximation points to determine if a segment should be subdivided.

- **segments** – minimum segment count

New in version 0.15.

**point**(t: float) → Vec3

Returns a point for parameter `t` in range [0, 1] as `Vec3` object.

**points**(t: Iterable[float]) → Iterable[Vec3]

Yields multiple points for parameters in vector `t` as `Vec3` objects. Parameters have to be in range [0, 1].
derivative \( (t: \text{float}) \rightarrow \text{Tuple}[\text{Vec3}, \text{Vec3}, \text{Vec3}] \)

Returns (point, 1st derivative, 2nd derivative) tuple for parameter \( t \) in range [0, 1] as \text{Vec3} objects.

derivatives \( (t: \text{Iterable[\text{float}]}) \rightarrow \text{Iterable[\text{Tuple}[\text{Vec3}, \text{Vec3}, \text{Vec3}]]} \)

Returns multiple (point, 1st derivative, 2nd derivative) tuples for parameter vector \( t \) as \text{Vec3} objects. Parameters in range [0, 1]

**Bezier4P**

class ezdxf.math.Bezier4P (\text{defpoints: Sequence[Vertex]})

Implements an optimized cubic Bézier curve for exact 4 control points.

A Bézier curve is a parametric curve, parameter \( t \) goes from 0 to 1, where 0 is the first control point and 1 is the fourth control point.

Special behavior:

- 2D control points in, returns 2D results as \text{Vec2} objects
- 3D control points in, returns 3D results as \text{Vec3} objects
- Object is immutable.

**Parameters** \text{defpoints} – iterable of definition points as \text{Vec2} or \text{Vec3} compatible objects.

control_points

Control points as tuple of \text{Vec3} or \text{Vec2} objects.

reverse () \rightarrow \text{Bezier4P}

Returns a new Bézier-curve with reversed control point order.

transform \( (m: \text{Matrix44}) \rightarrow \text{Bezier4P} \)

General transformation interface, returns a new \text{Bezier4P} curve and it is always a 3D curve.

**Parameters** \( m \) – 4x4 transformation matrix (ezdxf.math.Matrix44)

New in version 0.14.

approximate \( (\text{segments: int}) \rightarrow \text{Iterable[Union[\text{Vec3}, \text{Vec2}]]} \)

Approximate Bézier curve by vertices, yields \text{segments} + 1 vertices as \( (x, \ y[, \ z]) \) tuples.

**Parameters** \text{segments} – count of segments for approximation

flattening \( (\text{distance: float}, \text{segments: int}=4) \rightarrow \text{Iterable[\text{Union[\text{Vec3}, \text{Vec2}]}} \)

Adaptive recursive flattening. The argument \text{segments} is the minimum count of approximation segments, if the distance from the center of the approximation segment to the curve is bigger than \text{distance} the segment will be subdivided.

**Parameters**

- \text{distance} – maximum distance from the center of the cubic (C3) curve to the center of the linear (C1) curve between two approximation points to determine if a segment should be subdivided.
- \text{segments} – minimum segment count

New in version 0.15.

approximated_length \( (\text{segments: int} = 128) \rightarrow \text{float} \)

Returns estimated length of Bèzier-curve as approximation by line \text{segments}. 

338 Chapter 6. Contents
### Bezier3P

**class** `ezdxf.math.Bezier3P(defpoints: Sequence[Vertex])`

Implements an optimized quadratic Bézier curve for exact 3 control points.

**Special behavior:**
- 2D control points in, returns 2D results as `Vec2` objects
- 3D control points in, returns 3D results as `Vec3` objects
- Object is immutable.

**Parameters**
- `defpoints` – iterable of definition points as `Vec2` or `Vec3` compatible objects.

**control_points**
Control points as tuple of `Vec3` or `Vec2` objects.

**reverse()** → Bezier3P
Returns a new Bézier-curve with reversed control point order.

**transform(m: Matrix44)** → Bezier3P
General transformation interface, returns a new Bezier3P curve and it is always a 3D curve.

**Parameters**
- `m` – 4x4 transformation matrix (`ezdxf.math.Matrix44`)

**approximate(segments: int)** → Iterable[Union[Vec3, Vec2]]
Approximate Bézier curve by vertices, yields `segments` + 1 vertices as `(x, y [, z])` tuples.

**Parameters**
- `segments` – count of segments for approximation

**flattening(distance: float, segments: int=4)** → Iterable[Union[Vec3, Vec2]]
Adaptive recursive flattening. The argument `segments` is the minimum count of approximation segments, if the distance from the center of the approximation segment to the curve is bigger than `distance` the segment will be subdivided.

**Parameters**
- `distance` – maximum distance from the center of the quadratic (C2) curve to the center of the linear (C1) curve between two approximation points to determine if a segment should be subdivided.
- `segments` – minimum segment count

**approximated_length(segments: int = 128)** → float
Returns estimated length of Bézier-curve as approximation by line segments.

**point(t: float)** → Union[Vec3, Vec2]
Returns point for location `t` at the Bèzier-curve.

**Parameters**
- `t` – curve position in the range `[0, 1]`
tangent \( (t: \text{float}) \rightarrow \text{Union[Vec3, Vec2]} \)

Returns direction vector of tangent for location \( t \) at the Bézier-curve.

**Parameters**

\( t \) – curve position in the range \([0, 1]\)

### BezierSurface

**class** ezdxf.math.BezierSurface (defpoints: List[List[Vertex]])

**BezierSurface** defines a mesh of \( m \times n \) control points. This is a parametric surface, which means the \( m \)-dimension goes from \( 0 \) to \( 1 \) as parameter \( u \) and the \( n \)-dimension goes from \( 0 \) to \( 1 \) as parameter \( v \).

**Parameters**

\[ \text{defpoints} \] – matrix (list of lists) of \( m \) rows and \( n \) columns: \([ [m1n1, m1n2, \ldots], [m2n1, m2n2, \ldots] \ldots \] each element is a 3D location as \((x, y, z)\) tuple.

- **nrows**
  - count of rows (m-dimension)

- **ncols**
  - count of columns (n-dimension)

**point** \((u: \text{float}, v: \text{float}) \rightarrow \text{ezdxf.math._vector.Vec3}\)

Returns a point for location \((u, v)\) at the Bézier surface as \((x, y, z)\) tuple, parameters \( u \) and \( v \) in the range of \([0, 1]\).

**approximate** (usegs: int, vsegs: int) \rightarrow List[List[ezdxf.math._vector.Vec3]]

Approximate surface as grid of \((x, y, z)\) \text{Vec3}.

**Parameters**

- **usegs** – count of segments in \( u \)-direction (m-dimension)
- **vsegs** – count of segments in \( v \)-direction (n-dimension)

**Returns** list of usegs + 1 rows, each row is a list of vsegs + 1 vertices as \text{Vec3}.

### EulerSpiral

**class** ezdxf.math.EulerSpiral (curvature: float = 1.0)

This class represents an euler spiral (clothoid) for \textit{curvature} (Radius of curvature).

This is a parametric curve, which always starts at the origin \( = (0, 0) \).

**Parameters**

\[ \text{curvature} \] – radius of curvature

**radius** \((t: \text{float}) \rightarrow \text{float}\)

Get radius of circle at distance \( t \).

**tangent** \((t: \text{float}) \rightarrow \text{Vec3}\)

Get tangent at distance \( t \) as :class:`Vec3` object.

**distance** \((radius: \text{float}) \rightarrow \text{float}\)

Get distance L from origin for \textit{radius}.

**point** \((t: \text{float}) \rightarrow \text{Vec3}\)

Get point at distance \( t \) as :class:`Vec3`.

**circle_center** \((t: \text{float}) \rightarrow \text{Vec3}\)

Get circle center at distance \( t \).

*Changed in version 0.10: renamed from circle_midpoint*
approximate \((\text{length: float, segments: int}) \rightarrow \text{Iterable[Vec3]}\)
Approximate curve of length with line segments.
Generates segments+1 vertices as Vec3 objects.

bspline \((\text{length: float, segments: int = 10, degree: int = 3, method: str = 'uniform'}) \rightarrow \text{BSpline}\)
Approximate euler spiral as B-spline.

Parameters
- **length** – length of euler spiral
- **segments** – count of fit points for B-spline calculation
- **degree** – degree of BSpline
- **method** – calculation method for parameter vector \(t\)

Returns BSpline

### Linear Algebra

#### Functions

**ezdxf.math.gauss_jordan_solver** \((A: \text{Iterable[Iterable[float]]}, B: \text{Iterable[Iterable[float]]}) \rightarrow \text{Tuple[Matrix, Matrix]}\)
Solves the linear equation system given by a nxn Matrix \(A \cdot x = B\), right-hand side quantities as nxm Matrix B by the Gauss-Jordan algorithm, which is the slowest of all, but it is very reliable. Returns a copy of the modified input matrix \(A\) and the result matrix \(x\).
Internally used for matrix inverse calculation.

Parameters
- \(A\) – matrix \([[a11, a12, \ldots, a1n], [a21, a22, \ldots, a2n], \ldots [an1, an2, \ldots, ann]]\)
- \(B\) – matrix \([[b11, b12, \ldots, b1m], [b21, b22, \ldots, b2m], \ldots [bn1, bn2, \ldots, bnm]]\)

Returns 2-tuple of Matrix objects

Raises ZeroDivisionError – singular matrix

**ezdxf.math.gauss_jordan_inverse** \((A: \text{Iterable[Iterable[float]]}) \rightarrow \text{Matrix}\)
Returns the inverse of matrix \(A\) as Matrix object.

Hint: For small matrices \((n<10)\) is this function faster than LUDecomposition(m).inverse() and as fast even if the decomposition is already done.

Raises ZeroDivisionError – singular matrix

**ezdxf.math.gauss_vector_solver** \((A: \text{Iterable[Iterable[float]]}, B: \text{Iterable[float]}) \rightarrow \text{List[float]}\)
Solves the linear equation system given by a nxn Matrix \(A \cdot x = B\), right-hand side quantities as vector \(B\) with n elements by the Gauss-Elimination algorithm, which is faster than the Gauss-Jordan algorithm. The speed improvement is more significant for solving multiple right-hand side quantities as matrix at once.
Reference implementation for error checking.

Parameters
• **A** – matrix \([a_{11}, a_{12}, \ldots, a_{1n}], [a_{21}, a_{22}, \ldots, a_{2n}], [a_{21}, a_{22}, \ldots, a_{2n}], \ldots [a_{n1}, a_{n2}, \ldots, a_{nn}]\)

• **B** – vector \([b_1, b_2, \ldots, b_n]\)

**Returns** vector as list of floats

**Raises** ZeroDivisionError – singular matrix

{\texttt{ezdxf.math.gauss_matrix_solver}} (\textit{A}: Iterable[Iterable[float]], \textit{B}: Iterable[Iterable[float]]) \rightarrow Matrix

Solves the linear equation system given by a \(nxn\) Matrix \(A\). \(x = B\), right-hand side quantities as \(nxm\) Matrix \(B\) by the Gauss-Elimination algorithm, which is faster than the Gauss-Jordan algorithm.

Reference implementation for error checking.

**Parameters**

• **A** – matrix \([a_{11}, a_{12}, \ldots, a_{1n}], [a_{21}, a_{22}, \ldots, a_{2n}], [a_{21}, a_{22}, \ldots, a_{2n}], \ldots [a_{n1}, a_{n2}, \ldots, a_{nn}]\)

• **B** – matrix \([b_{11}, b_{12}, \ldots, b_{1m}], [b_{21}, b_{22}, \ldots, b_{2m}], \ldots [b_{n1}, b_{n2}, \ldots, b_{nm}]\)

**Returns** matrix as \texttt{Matrix} object

**Raises** ZeroDivisionError – singular matrix

{\texttt{ezdxf.math.tridiagonal_vector_solver}} (\textit{A}: Iterable[Iterable[float]], \textit{B}: Iterable[float]) \rightarrow List[float]

Solves the linear equation system given by a tri-diagonal \(nxn\) Matrix \(A\). \(x = B\), right-hand side quantities as vector \(B\). Matrix \(A\) is diagonal matrix defined by 3 diagonals \([-1 (a), 0 (b), +1 (c)]\).

Note: a0 is not used but has to be present, cn-1 is also not used and must not be present.

If an ZeroDivisionError exception occurs, the equation system can possibly be solved by \texttt{BandedMatrixLU(A, 1, 1).solve_vector(B)}

**Parameters**

• **A** – diagonal matrix \([a_{0..an-1}], [b_{0..bn-1}], [c_{0..cn-1}]\)

\[
\begin{bmatrix}
b_0, c_0, 0, 0, \ldots, \\
a_1, b_1, c_1, 0, \ldots, \\
0, a_2, b_2, c_2, \ldots, \\
\ldots 
\end{bmatrix}
\]

• **B** – iterable of floats \([b_1, b_1, \ldots, b_n]\)

**Returns** list of floats

**Raises** ZeroDivisionError – singular matrix

{\texttt{ezdxf.math.tridiagonal_matrix_solver}} (\textit{A}: Iterable[Iterable[float]], \textit{B}: Iterable[Iterable[float]]) \rightarrow Matrix

Solves the linear equation system given by a tri-diagonal \(nxn\) Matrix \(A\). \(x = B\), right-hand side quantities as \(nxm\) Matrix \(B\). Matrix \(A\) is diagonal matrix defined by 3 diagonals \([-1 (a), 0 (b), +1 (c)]\).

Note: a0 is not used but has to be present, cn-1 is also not used and must not be present.

If an ZeroDivisionError exception occurs, the equation system can possibly be solved by \texttt{BandedMatrixLU(A, 1, 1).solve_vector(B)}

**Parameters**

• **A** – diagonal matrix \([a_{0..an-1}], [b_{0..bn-1}], [c_{0..cn-1}]\)
ezdxf Documentation, Release 0.16.2

| B – matrix [[b11, b12, ..., b1m], [b21, b22, ..., b2m], ... [bn1, bn2, ..., bnm]] |

Returns matrix as \texttt{Matrix} object

Raises \texttt{ZeroDivisionError} – singular matrix

\texttt{ezdxf.math.banded_matrix}(A: \texttt{Matrix}, check\_all=True) → Tuple[int, int]

Transform matrix \texttt{A} into a compact banded matrix representation. Returns compact representation as \texttt{Matrix} object and lower- and upper band count \texttt{m1} and \texttt{m2}.

Parameters

• \texttt{A} – input \texttt{Matrix}

• \texttt{check\_all} – check all diagonals if \texttt{True} or abort testing after first all zero diagonal if \texttt{False}.

\texttt{ezdxf.math.detect_banded_matrix}(A: \texttt{Matrix}, check\_all=True) → Tuple[int, int]

Returns lower- and upper band count \texttt{m1} and \texttt{m2}.

Parameters

• \texttt{A} – input \texttt{Matrix}

• \texttt{check\_all} – check all diagonals if \texttt{True} or abort testing after first all zero diagonal if \texttt{False}.

\texttt{ezdxf.math.compact_banded_matrix}(A: \texttt{Matrix}, m1: int, m2: int) → \texttt{Matrix}

Returns compact banded matrix representation as \texttt{Matrix} object.

Parameters

• \texttt{A} – matrix to transform

• \texttt{m1} – lower band count, excluding main matrix diagonal

• \texttt{m2} – upper band count, excluding main matrix diagonal

\texttt{ezdxf.math.freeze_matrix}(A: Union[\texttt{MatrixData}, \texttt{Matrix}]) → \texttt{Matrix}

Returns a frozen matrix, all data is stored in immutable tuples.

Matrix Class

\texttt{class ezdxf.math.Matrix}(items: \texttt{Any} = \texttt{None}, shape: \texttt{Tuple[int, int]} = \texttt{None}, matrix: \texttt{List[List[float]]} = \texttt{None})

Basic matrix implementation without any optimization for speed of memory usage. Matrix data is stored in row major order, this means in a list of rows, where each row is a list of floats. Direct access to the data is accessible by the attribute \texttt{Matrix.matrix}.

The matrix can be frozen by function \texttt{freeze\_matrix()} or method \texttt{Matrix.freeze()}, than the data is stored in immutable tuples.

Initialization:

• \texttt{Matrix(shape=(rows, cols))} … new matrix filled with zeros

• \texttt{Matrix(matrix[, shape=(rows, cols)])} … from copy of matrix and optional reshape
• Matrix([[row_0], [row_1], ..., [row_n]]) ... from Iterable[Iterable[float]]

• Matrix([a1, a2, ..., an], shape=(rows, cols)) ... from Iterable[float] and shape

nrows
Count of matrix rows.

ncols
Count of matrix columns.

shape
Shape of matrix as (n, m) tuple for n rows and m columns.

static reshape(items: Iterable[float], shape: Tuple[int, int]) -> ezdxf.math.linalg.Matrix
Returns a new matrix for iterable items in the configuration of shape.

classmethod identity(shape: Tuple[int, int]) -> ezdxf.math.linalg.Matrix
Returns the identity matrix for configuration shape.

row(index) -> List[float]
Returns row index as list of floats.

iter_row(index) -> Iterable[float]
Yield values of row index.

col(index) -> List[float]
Return column index as list of floats.

iter_col(index) -> Iterable[float]
Yield values of column index.

diag(index) -> List[float]
Returns diagonal index as list of floats.

An index of 0 specifies the main diagonal, negative values specifies diagonals below the main diagonal and positive values specifies diagonals above the main diagonal.

e.g. given a 4x4 matrix:

• index 0 is [00, 11, 22, 33],
• index -1 is [10, 21, 32] and
• index +1 is [01, 12, 23]

iter_diag(index) -> Iterable[float]
Yield values of diagonal index, see also diag().

rows() -> List[List[float]]
Return a list of all rows.

cols() -> List[List[float]]
Return a list of all columns.

set_row(index: int, items: Union[float, Iterable[float]] = 1.0) -> None
Set row values to a fixed value or from an iterable of floats.

set_col(index: int, items: Union[float, Iterable[float]] = 1.0) -> None
Set column values to a fixed value or from an iterable of floats.

set_diag(index: int = 0, items: Union[float, Iterable[float]] = 1.0) -> None
Set diagonal values to a fixed value or from an iterable of floats.

An index of 0 specifies the main diagonal, negative values specifies diagonals below the main diagonal and positive values specifies diagonals above the main diagonal.
e.g. given a 4x4 matrix: index 0 is [00, 11, 22, 33], index -1 is [10, 21, 32] and index +1 is [01, 12, 23]

append_row(items: Sequence[Float]) → None
Append a row to the matrix.

append_col(items: Sequence[Float]) → None
Append a column to the matrix.

swap_rows(a: int, b: int) → None
Swap rows a and b inplace.

swap_cols(a: int, b: int) → None
Swap columns a and b inplace.

transpose() → Matrix
Returns a new transposed matrix.

inverse() → Matrix
Returns inverse of matrix as new object.

determinant() → float
Returns determinant of matrix, raises ZeroDivisionError if matrix is singular.

freeze() → Matrix
Returns a frozen matrix, all data is stored in immutable tuples.

lu_decomp() → LUDecomposition
Returns the LU decomposition as LUDecomposition object, a faster linear equation solver.

getitem(item: Tuple[int, int]) → float
Get value by (row, col) index tuple, fancy slicing as known from numpy is not supported.

setitem(item: Tuple[int, int], value: float)
Set value by (row, col) index tuple, fancy slicing as known from numpy is not supported.

eq(other: Matrix) → bool
Returns True if matrices are equal, tolerance value for comparison is adjustable by the attribute Matrix.abs_tol.

add(other: Union[Matrix, Float]) → Matrix
Matrix addition by another matrix or a float, returns a new matrix.

sub(other: Union[Matrix, Float]) → Matrix
Matrix subtraction by another matrix or a float, returns a new matrix.

mul(other: Union[Matrix, Float]) → Matrix
Matrix multiplication by another matrix or a float, returns a new matrix.

LUDecomposition Class

class ezdxf.math.LUDecomposition(A: Iterable[Iterable[Float]])
Represents a LU decomposition matrix of A, raise ZeroDivisionError for a singular matrix.

This algorithm is a little bit faster than the Gauss-Elimination algorithm using CPython and much faster when using pypy.

The LUDecomposition.matrix attribute gives access to the matrix data as list of rows like in the Matrix class, and the LUDecomposition.index attribute gives access to the swapped row indices.

Parameters A – matrix [[a11, a12, ..., a1n], [a21, a22, ..., a2n], [a21, a22, ..., a2n], ... [an1, an2, ..., ann]]
Raises ZeroDivisionError – singular matrix

nrows
Count of matrix rows (and cols).

solve_vector \((B: \text{Iterable}[\text{float}]) \rightarrow \text{List}[\text{float}]\)
Solves the linear equation system given by the nxn Matrix \(A \cdot x = B\), right-hand side quantities as vector \(B\) with \(n\) elements.

Parameters \(B\) – vector \([b_1, b_2, \ldots, b_n]\)

Returns vector as list of floats

solve_matrix \((B: \text{Iterable}[\text{Iterable}[\text{float}]]) \rightarrow \text{Matrix}\)
Solves the linear equation system given by the nxn Matrix \(A \cdot x = B\), right-hand side quantities as nxm Matrix \(B\).

Parameters \(B\) – matrix \([[b_{11}, b_{12}, \ldots, b_{1m}], [b_{21}, b_{22}, \ldots, b_{2m}], \ldots, [b_{n1}, b_{n2}, \ldots, b_{nm}]\])

Returns matrix as Matrix object

inverse () \(\rightarrow\) Matrix
Returns the inverse of matrix as Matrix object, raise ZeroDivisionError for a singular matrix.

determinant () \(\rightarrow\) float
Returns the determinant of matrix, raises ZeroDivisionError if matrix is singular.

BandedMatrixLU Class

class ezdxf.math.BandedMatrixLU \((A: \text{ezdxf.math.linalg.Matrix}, m1: \text{int}, m2: \text{int})\)
Represents a LU decomposition of a compact banded matrix.

upper
Upper triangle

lower
Lower triangle

m1
Lower band count, excluding main matrix diagonal

m2
Upper band count, excluding main matrix diagonal

index
Swapped indices

nrows
Count of matrix rows.

solve_vector \((B: \text{Iterable}[\text{float}]) \rightarrow \text{List}[\text{float}]\)
Solves the linear equation system given by the banded nxn Matrix \(A \cdot x = B\), right-hand side quantities as vector \(B\) with \(n\) elements.

Parameters \(B\) – vector \([b_1, b_2, \ldots, b_n]\)

Returns vector as list of floats

solve_matrix \((B: \text{Iterable}[\text{Iterable}[\text{float}]]) \rightarrow \text{Matrix}\)
Solves the linear equation system given by the banded nxn Matrix \(A \cdot x = B\), right-hand side quantities as nxm Matrix \(B\).

Parameters \(B\) – matrix \([[b_{11}, b_{12}, \ldots, b_{1m}], [b_{21}, b_{22}, \ldots, b_{2m}], \ldots, [b_{n1}, b_{n2}, \ldots, b_{nm}]\])
Returns matrix as `Matrix` object

**determinant** () → float

Returns the determinant of matrix.

### 6.8.5 Construction

**Path**

This module implements a geometric `Path`, supported by several render backends, with the goal to create such paths from DXF entities like LWPOLYLINE, POLYLINE or HATCH and send them to the render backend, see `ezdxf.addons.drawing`.

Minimum common interface:

- **matplotlib: `PathPatch`**
  - matplotlib.path.Path() codes:
    - MOVETO
    - LINETO
    - CURVE3 - quadratic Bèzier-curve
    - CURVE4 - cubic Bèzier-curve

- **PyQt: `QPainterPath`**
  - moveTo()
  - lineTo()
  - quadTo() - quadratic Bèzier-curve (converted to a cubic Bèzier-curve)
  - cubicTo() - cubic Bèzier-curve

- **PyCairo: `Context`**
  - move_to()
  - line_to()
  - no support for quadratic Bèzier-curve
  - curve_to() - cubic Bèzier-curve

- **SVG: `SVG-Path`**
  - “M” - absolute move to
  - “L” - absolute line to
  - “Q” - absolute quadratic Bèzier-curve
  - “C” - absolute cubic Bèzier-curve

ARC and ELLIPSE entities are approximated by multiple cubic Bézier-curves, which are close enough for display rendering. Non-rational SPLINES of 3rd degree can be represented exact as multiple cubic Bézier-curves, other B-splines will be approximated. The XLINE and the RAY entities are not supported, because of their infinite nature.

This `Path` class is a full featured 3D object, although the backends only support 2D paths.

---

**Hint:** A `Path` can not represent a point. A `Path` with only a start point yields no vertices!
Changed in version 0.16: Refactored the module `ezdxf.render.path` into the subpackage `ezdxf.path`.

The usability of the `Path` class expanded by the introduction of the reverse conversion from `Path` to DXF entities (LWPOLYLINE, POLYLINE, LINE), and many other tools in `ezdf` v0.16. To emphasize this new usability, the `Path` class has got its own subpackage `ezdxf.path`.

### Warning

Always import from the top level `ezdxf.path`, never from the sub-modules

#### Factory Functions

Functions to create `Path` objects from other objects.

**ezdxf.path.make_path** *(entity: DXFEntity) → Path*

Factory function to create a single `Path` object from a DXF entity. Supported DXF types:

- LINE
- CIRCLE
- ARC
- ELLIPSE
- SPLINE and HELIX
- LWPOLYLINE
- 2D and 3D POLYLINE
- SOLID, TRACE, 3DFACE
- IMAGE, WIPEOUT clipping path
- VIEWPORT clipping path

The HATCH entity consist of multiple boundary paths and is not convertible into a single `Path` object and therefore not supported by this function.

**Parameters**

- **entity** – DXF entity
- **segments** – minimal count of cubic Bézier-curves for elliptical arcs like CIRCLE, ARC, ELLIPSE, see `Path.add_ellipse()`
- **level** – subdivide level for SPLINE approximation, see `Path.add_spline()`

**Raises** `TypeError` – for unsupported DXF types

New in version 0.16.

**ezdxf.path.from_hatch** *(hatch: Hatch) → Iterable[Path]*

Yield all HATCH boundary paths as separated `Path` objects.

New in version 0.16.

**ezdxf.path.from_vertices** *(vertices: Iterable[Vertex], close=False) → Path*

Returns a `Path` object from the given `vertices`.

**ezdxf.path.from_matplotlib_path** *(mpath, curves=True) → Iterable[Path]*

Yields multiple `Path` objects from a Matplotlib `Path` (TextPath) object. (requires Matplotlib)

New in version 0.16.
ezdxf.path.from_qpainter_path(qpath) → Iterable[Path]
Yields multiple Path objects from a QPainterPath. (requires PyQt5)
New in version 0.16.

**Render Functions**

Functions to create DXF entities from paths and add them to the modelspace, a paperspace layout or a block definition.

ezdxf.path.render_lwpolylines(layout: Layout, paths: Iterable[Path], *, distance: float = 0.01, segments: int = 4, extrusion: Vertex = (0, 0, 1), dfattribs: Dict = None) → EntityQuery
Render the given paths into layout as LWPolyline entities. The extrusion vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

**Parameters**
- layout – the modelspace, a paperspace layout or a block definition
- paths – iterable of Path objects
- distance – maximum distance, see Path.flattening()
- segments – minimum segment count per Bézier curve
- extrusion – extrusion vector for all paths
- dfattribs – additional DXF attribs

**Returns** created entities in an EntityQuery object
New in version 0.16.

ezdxf.path.render_polylines2d(layout: Layout, paths: Iterable[Path], *, distance: float = 0.01, segments: int = 4, extrusion: Vertex = (0, 0, 1), dfattribs: Dict = None) → EntityQuery
Render the given paths into layout as 2D Polyline entities. The extrusion vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

**Parameters**
- layout – the modelspace, a paperspace layout or a block definition
- paths – iterable of Path objects
- distance – maximum distance, see Path.flattening()
- segments – minimum segment count per Bézier curve
- extrusion – extrusion vector for all paths
- dfattribs – additional DXF attribs

**Returns** created entities in an EntityQuery object
New in version 0.16.

ezdxf.path.render_hatches(layout: Layout, paths: Iterable[Path], *, edge_path = True, distance: float = 0.01, segments: int = 4, g1_tol: float = 1e-4, extrusion: Vertex = (0, 0, 1), dfattribs: Dict = None) → EntityQuery
Render the given paths into layout as Hatch entities. The extrusion vector is applied to all paths, all vertices
are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

**Parameters**
- `layout` – the modelspace, a paperspace layout or a block definition
- `paths` – iterable of `Path` objects
- `edge_path` – True for edge paths build of LINE and SPLINE edges, False for only LWPOLYLINE paths as boundary paths
- `distance` – maximum distance, see `Path.flattening()`
- `segments` – minimum segment count per Bézier curve to flatten LWPOLYLINE paths
- `g1_tol` – tolerance for G1 continuity check to separate SPLINE edges
- `extrusion` – extrusion vector for all paths
- `dxfattribs` – additional DXF attribs

**Returns** created entities in an `EntityQuery` object

New in version 0.16.

```python
ezdxf.path.render_polylines3d(layout: Layout, paths: Iterable[Path], *, distance: float = 0.01, segments: int = 4, dxfattribs: Dict = None) → EntityQuery
```

Render the given `paths` into `layout` as 3D `Polyline` entities.

**Parameters**
- `layout` – the modelspace, a paperspace layout or a block definition
- `paths` – iterable of `Path` objects
- `distance` – maximum distance, see `Path.flattening()`
- `segments` – minimum segment count per Bézier curve
- `dxfattribs` – additional DXF attribs

**Returns** created entities in an `EntityQuery` object

New in version 0.16.

```python
ezdxf.path.render_lines(layout: Layout, paths: Iterable[Path], *, distance: float = 0.01, segments: int = 4, dxfattribs: Dict = None) → EntityQuery
```

Render the given `paths` into `layout` as `Line` entities.

**Parameters**
- `layout` – the modelspace, a paperspace layout or a block definition
- `paths` – iterable of `Path` objects
- `distance` – maximum distance, see `Path.flattening()`
- `segments` – minimum segment count per Bézier curve
- `dxfattribs` – additional DXF attribs

**Returns** created entities in an `EntityQuery` object

New in version 0.16.
ezdxf.path.render_splines_and_polylines(layout: Layout, paths: Iterable[Path], *, g1_tol: float = 1e-4, dxfattribs: Dict = None) → EntityQuery

Render the given paths into layout as Spline and 3D Polyline entities.

**Parameters**

- `layout` – the modelspace, a paperspace layout or a block definition
- `paths` – iterable of Path objects
- `g1_tol` – tolerance for G1 continuity check
- `dxfattribs` – additional DXF attribs

**Returns** created entities in an EntityQuery object

New in version 0.16.

**Entity Maker**

Functions to create DXF entities from paths.

ezdxf.path.to_lwpolylines(paths: Iterable[Path], *, distance: float = 0.01, segments: int = 4, extrusion: Vertex = (0, 0, 1), dxfattribs: Dict = None) → Iterable[LWPolyline]

Convert the given paths into LWPolyline entities. The extrusion vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

**Parameters**

- `paths` – iterable of Path objects
- `distance` – maximum distance, see Path.flattening()
- `segments` – minimum segment count per Bézier curve
- `extrusion` – extrusion vector for all paths
- `dxfattribs` – additional DXF attribs

**Returns** iterable of LWPolyline objects

New in version 0.16.

ezdxf.path.to_polylines2d(paths: Iterable[Path], *, distance: float = 0.01, segments: int = 4, extrusion: Vertex = (0, 0, 1), dxfattribs: Dict = None) → Iterable[Polyline]

Convert the given paths into 2D Polyline entities. The extrusion vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

**Parameters**

- `paths` – iterable of Path objects
- `distance` – maximum distance, see Path.flattening()
- `segments` – minimum segment count per Bézier curve
- `extrusion` – extrusion vector for all paths
- `dxfattribs` – additional DXF attribs

**Returns** iterable of 2D Polyline objects
New in version 0.16.

**ezdxf.path.to_hatches** *(paths: Iterable[Path], *, edge_path: True, distance: float = 0.01, segments: int = 4, g1_tol: float = 1e-4, extrusion: Vertex = (0, 0, 1), dxfattribs: Dict = None) → Iterable[Hatch]*

Convert the given *paths* into *Hatch* entities. Uses LWPOLYLINE paths for boundaries without curves and edge paths, build of LINE and SPLINE edges, as boundary paths for boundaries including curves. The *extrusion* vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

**Parameters**
- **paths** – iterable of *Path* objects
- **edge_path** – *True* for edge paths build of LINE and SPLINE edges, *False* for only LWPOLYLINE paths as boundary paths
- **distance** – maximum distance, see *Path.flattening()*
- **segments** – minimum segment count per Bézier curve to flatten LWPOLYLINE paths
- **g1_tol** – tolerance for G1 continuity check to separate SPLINE edges
- **extrusion** – extrusion vector to all paths
- **dxfattribs** – additional DXF attribs

**Returns** iterable of *Hatch* objects

New in version 0.16.

**ezdxf.path.to_polylines3d** *(paths: Iterable[Path], *, distance: float = 0.01, segments: int = 4, dxfattribs: Dict = None) → Iterable[Polyline]*

Convert the given *paths* into 3D *Polyline* entities.

**Parameters**
- **paths** – iterable of *Path* objects
- **distance** – maximum distance, see *Path.flattening()*
- **segments** – minimum segment count per Bézier curve
- **dxfattribs** – additional DXF attribs

**Returns** iterable of 3D *Polyline* objects

New in version 0.16.

**ezdxf.path.to_lines** *(paths: Iterable[Path], *, distance: float = 0.01, segments: int = 4, dxfattribs: Dict = None) → Iterable[Line]*

Convert the given *paths* into *Line* entities.

**Parameters**
- **paths** – iterable of *Path* objects
- **distance** – maximum distance, see *Path.flattening()*
- **segments** – minimum segment count per Bézier curve
- **dxfattribs** – additional DXF attribs

**Returns** iterable of *Line* objects

New in version 0.16.
ezdxf.path.to_splines_and_polylines(paths: Iterable[Path], *, g1_tol: float = 1e-4, dxfattribs: Dict = None) → Iterable[Union[Spline, Polyline]]

Convert the given paths into Spline and 3D Polyline entities.

**Parameters**
- `paths` – iterable of `Path` objects
- `g1_tol` – tolerance for G1 continuity check
- `dxfattribs` – additional DXF attribs

**Returns** iterable of `Line` objects

New in version 0.16.

tool Maker

Functions to create construction tools.

ezdxf.path.to_bsplines_and_vertices(path: Path, g1_tol: float = 1e-4) → Iterable[Union[BSpline, List[Vec3]]]

Convert a `Path` object into multiple cubic B-splines and polylines as lists of vertices. Breaks adjacent Bèzier without G1 continuity into separated B-splines.

**Parameters**
- `path` – `Path` objects
- `g1_tol` – tolerance for G1 continuity check

**Returns** `BSpline` and lists of `Vec3`

New in version 0.16.

ezdxf.path.to_matplotlib_path(paths: Iterable[Path], extrusion: (0, 0, 1)) → matplotlib.path.Path

Convert the given paths into a single `matplotlib.path.Path` object. The extrusion vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The Matplotlib `Path` is a 2D object with OCS coordinates and the z-elevation is lost. (requires Matplotlib)

**Parameters**
- `paths` – iterable of `Path` objects
- `extrusion` – extrusion vector for all paths

**Returns** `matplolib Path` in OCS!

New in version 0.16.

ezdxf.path.to_qpainter_path(paths: Iterable[Path], extrusion: (0, 0, 1)) → PyQt5.QtGui.QPainterPath

Convert the given paths into a `PyQt5.QtGui.QPainterPath` object. The extrusion vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The QPainterPath is a 2D object with OCS coordinates and the z-elevation is lost. (requires PyQt5)

**Parameters**
- `paths` – iterable of `Path` objects
- `extrusion` – extrusion vector for all paths
Returns `QPainterPath` in OCS!

New in version 0.16.

Utility Functions

```python
ezdxf.path.transform_paths(paths: Iterable[Path], m: Matrix44) → List[Path]
```

Transform multiple `Path` objects at once by transformation matrix `m`. Returns a list of the transformed `Path` objects.

Parameters

- **paths** – iterable of `Path` objects
- **m** – transformation matrix of type `Matrix44`

```python
ezdxf.path.transform_paths_to_ocs(paths: Iterable[Path], ocs: OCS) → List[Path]
```

Transform multiple `Path` objects at once from WCS to OCS. Returns a list of the transformed `Path` objects.

Parameters

- **paths** – iterable of `Path` objects
- **ocs** – OCS transformation of type `OCS`

```python
ezdxf.path.bbox(paths: Iterable[Path]) → BoundingBox
```

Returns the `BoundingBox` for the given paths.

Parameters

- **paths** – iterable of `Path` objects
- **flatten** – value != 0 for bounding box calculation from the flattened
- **and value == 0 for bounding box from the control vertices.**
- **(path)** –
- **value is 0.01 as max flattening distance.**
- **(Default)** –
- **segments** – minimal segment count for flattening

```python
ezdxf.path.fit_paths_into_box(paths: Iterable[Path], size: Tuple[float, float, float], uniform = True, source_box: BoundingBox = None) → List[Path]
```

Scale the given `paths` to fit into a box of the given `size`, so that all path vertices are inside this borders. If `source_box` is `None` the default source bounding box is calculated from the control points of the `paths`.

Note: if the target size has a z-size of 0, the `paths` are projected into the xy-plane, same is true for the x-size, projects into the yz-plane and the y-size, projects into and xz-plane.

Parameters

- **paths** – iterable of `Path` objects
- **size** – target box size as tuple of x-, y- ond z-size values
- **uniform** – `True` for uniform scaling
- **source_box** – pass precalculated source bounding box, or `None` to calculate the default source bounding box from the control vertices

```python
ezdxf.path.add_bezier3p(path: Path, curves: Iterable[Bezier3P])
```

Add multiple quadratic Bèzier-curves to the given `path`.

Auto-detect the connection point to the given `path`, if neither the start- nor the end point of the curves is close to the path end point, a line from the path end point to the start point of the first curve will be added automatically.
Changed in version 0.16.2: add linear Bézier curve segments as LINE_TO commands

```python
ezdxf.path.add_bezier4p(path: Path, curves: Iterable[Bezier4P])
```
Add multiple cubic Bézier-curves to the given path.

Auto-detect the connection point to the given path, if neither the start- nor the end point of the curves is close to the path end point, a line from the path end point to the start point of the first curve will be added automatically.

Changed in version 0.16.2: add linear Bézier curve segments as LINE_TO commands

```python
ezdxf.path.add_ellipse(path: Path, ellipse: ConstructionEllipse, segments=1)
```
Add an elliptical arc as multiple cubic Bèzier-curves to the given path, use `from_arc()` constructor of class `ConstructionEllipse` to add circular arcs.

Auto-detect the connection point to the given path, if neither the start- nor the end point of the ellipse is close to the path end point, a line from the path end point to the ellipse start point will be added automatically (see `add_bezier4p()`).

By default the start of an empty path is set to the start point of the ellipse, setting argument `reset` to `False` prevents this behavior.

Parameters

- **path** – `Path` object
- **ellipse** – ellipse parameters as `ConstructionEllipse` object
- **segments** – count of Bèzier-curve segments, at least one segment for each quarter \( \pi/2 \), 1 for as few as possible.
- **reset** – set start point to start of ellipse if path is empty

```python
ezdxf.path.add_spline(path: Path, spline: BSpline, level=4)
```
Add a B-spline as multiple cubic Bèzier-curves.

Non-rational B-splines of 3rd degree gets a perfect conversion to cubic bezier curves with a minimal count of curve segments, all other B-spline require much more curve segments for approximation.

Auto-detect the connection point to the given path, if neither the start- nor the end point of the B-spline is close to the path end point, a line from the path end point to the start point of the B-spline will be added automatically. (see `add_bezier4p()`).

By default the start of an empty path is set to the start point of the spline, setting argument `reset` to `False` prevents this behavior.

Parameters

- **path** – `Path` object
- **spline** – B-spline parameters as `BSpline` object
- **level** – subdivision level of approximation segments
- **reset** – set start point to start of spline if path is empty

**Basic Shapes**

```python
ezdxf.path.unit_circle(start_angle: float = 0, end_angle: float = 2\pi, segments: int = 1, transform: Matrix44 = None) → Path
```
Returns an unit circle as a `Path` object, with the center at \((0, 0, 0)\) and the radius of 1 drawing unit.

The arc spans from the start- to the end angle in counter clockwise orientation. The end angle has to be greater than the start angle and the angle span has to be greater than 0.
ezdxf Documentation, Release 0.16.2

Parameters

- **start_angle** – start angle in radians
- **end_angle** – end angle in radians (end_angle > start_angle!)
- **segments** – count of Bèzier-curve segments, default is one segment for each arc quarter \(\frac{\pi}{2}\)
- **transform** – transformation Matrix applied to the unit circle

```python
ezdxf.path.wedge(start_angle: float, end_angle: float, segments: int = 1, transform: Matrix44 = None) → Path
```

Returns a wedge as a Path object, with the center at (0, 0, 0) and the radius of 1 drawing unit.

The arc spans from the start- to the end angle in counter clockwise orientation. The end angle has to be greater than the start angle and the angle span has to be greater than 0.

Parameters

- **start_angle** – start angle in radians
- **end_angle** – end angle in radians (end_angle > start_angle!)
- **segments** – count of Bèzier-curve segments, default is one segment for each arc quarter \(\frac{\pi}{2}\)
- **transform** – transformation Matrix applied to the wedge

```python
ezdxf.path.elliptic_transformation(center: Vertex = (0, 0, 0), radius: float = 1, ratio: float = 1, rotation: float = 0) → Matrix44
```

Returns the transformation matrix to transform an unit circle into an arbitrary circular- or elliptic arc.

Example how to create an ellipse with an major axis length of 3, a minor axis length 1.5 and rotated about 90°:

```python
m = elliptic_transformation(radius=3, ratio=0.5, rotation=math.pi / 2)
ellipse = shapes.unit_circle(transform=m)
```

Parameters

- **center** – curve center in WCS
- **radius** – radius of the major axis in drawing units
- **ratio** – ratio of minor axis to major axis
- **rotation** – rotation angle about the z-axis in radians

```python
ezdxf.path.rect(width: float = 1, height: float = 1, transform: Matrix44 = None) → Path
```

Returns a closed rectangle as a Path object, with the center at (0, 0, 0) and the given width and height in drawing units.

Parameters

- **width** – width of the rectangle in drawing units, width > 0
- **height** – height of the rectangle in drawing units, height > 0
- **transform** – transformation Matrix applied to the rectangle

```python
ezdxf.path.ngon(count: int, length: float = None, radius: float = 1.0, transform: Matrix44 = None) → Path
```

Returns a regular polygon a Path object, with the center at (0, 0, 0). The polygon size is determined by the edge length or the circum radius argument. If both are given length has higher priority. Default size is a radius of 1. The ngon starts with the first vertex is on the x-axis! The base geometry is created by function `ezdxf.render.forms.ngon()`.
Parameters

- **count** – count of polygon corners \( \geq 3 \)
- **length** – length of polygon side
- **radius** – circum radius, default is 1
- **transform** – transformation Matrix applied to the n gon

```python
ezdxf.path.star(count: int, r1: float, r2: float, transform: Matrix44 = None) ➞ Path
```

Returns a star shape as a `Path` object, with the center at (0, 0, 0).

Argument `count` defines the count of star spikes, `r1` defines the radius of the “outer” vertices and `r2` defines the radius of the “inner” vertices, but this does not mean that `r1` has to be greater than `r2`. The star shape starts with the first vertex is on the x-axis! The base geometry is created by function `ezdxf.render.forms.star()`.

Parameters

- **count** – spike count \( \geq 3 \)
- **r1** – radius 1
- **r2** – radius 2
- **transform** – transformation Matrix applied to the star

```python
ezdxf.path.gear(count: int, top_width: float, bottom_width: float, height: float, outside_radius: float, transform: Matrix44 = None) ➞ Path
```

Returns a gear (cogwheel) shape as a `Path` object, with the center at (0, 0, 0). The base geometry is created by function `ezdxf.render.forms.gear()`.

**Warning:** This function does not create correct gears for mechanical engineering!

Parameters

- **count** – teeth count \( \geq 3 \)
- **top_width** – teeth width at outside radius
- **bottom_width** – teeth width at base radius
- **height** – teeth height; base radius = outside radius - height
- **outside_radius** – outside radius
- **transform** – transformation Matrix applied to the gear shape

The `text2path` add-on provides additional functions to create paths from text strings and DXF text entities.

The **Path Class**

```python
class ezdxf.path.Path
```

- **start** `Path` start point, resetting the start point of an empty path is possible.
- **end** `Path` end point.
**is_closed**
Returns `True` if the start point is close to the end point.

**has_lines**
Returns `True` if the path has any line segments.

**has_curves**
Returns `True` if the path has any curve segments.

**control_vertices()**
Yields all path control vertices in consecutive order.

**has_clockwise_orientation() → bool**
Returns `True` if 2D path has clockwise orientation, ignores z-axis of all control vertices.

**line_to(location: Vec3)**
Add a line from actual path end point to location.

**curve3_to(location: Vec3, ctrl: Vec3)**
Add a quadratic Bèzier-curve from actual path end point to location, ctrl is the control point for the quadratic Bèzier-curve.

**curve4_to(location: Vec3, ctrl1: Vec3, ctrl2: Vec3)**
Add a cubic Bèzier-curve from actual path end point to location, ctrl1 and ctrl2 are the control points for the cubic Bèzier-curve.

**close() → None**
Close path by adding a line segment from the end point to the start point.

**clone() → Path**
Returns a new copy of `Path` with shared immutable data.

**reversed() → Path**
Returns a new `Path` with reversed segments and control vertices.

**clockwise() → Path**
Returns new `Path` in clockwise orientation.

**counter_clockwise() → Path**
Returns new `Path` in counter-clockwise orientation.

**transform(m: Matrix44) → Path**
Returns a new transformed path.

**approximate(segments: int=20) → Iterable[Vec3]**
Approximate path by vertices, `segments` is the count of approximation segments for each Bézier curve.

Does not yield any vertices for empty paths, where only a start point is present!

**flattening(distance: float, segments: int=16) → Iterable[Vec3]**
Approximate path by vertices and use adaptive recursive flattening to approximate Bèzier curves. The argument `segments` is the minimum count of approximation segments for each curve, if the distance from the center of the approximation segment to the curve is bigger than `distance` the segment will be subdivided.

Does not yield any vertices for empty paths, where only a start point is present!

**Parameters**
- `distance` – maximum distance from the center of the curve to the center of the line segment between two approximation points to determine if a segment should be subdivided.
- `segments` – minimum segment count per Bézier curve
Disassemble

New in version 0.16. This module provides tools for the recursive decomposition of nested block reference structures into a flat stream of DXF entities and converting DXF entities into geometric primitives of Path and MeshBuilder objects encapsulated into intermediate Primitive classes.

The Hatch entity is special because this entity can not be reduced into a single geometric primitive. The make_primitive() function returns an empty primitive, instead use the to_primitives() function to convert a Hatch entity into multiple (boundary path) primitives:

```python
primitives = list(to_primitives([hatch_entity]))
```

**Warning:** Do not expect advanced vectorization capabilities: Text entities like TEXT, ATTRIB, ATTDEF and MTEXT get only a rough border box representation. The text2path add-on can convert text into paths. VIEWPORT, IMAGE and WIPEOUT are represented by their clipping path. Unsupported entities: all ACIS based entities, XREF, UNDERLAY, ACAD_TABLE, RAY, XLINE. Unsupported entities will be ignored.

Text Boundary Calculation

Text boundary calculations are based on monospaced (fixed-pitch, fixed-width, non-proportional) font metrics, which do not provide a good accuracy for text height calculation and much less accuracy for text width calculation.

It is possible to improve this results by using font support from the Matplotlib package, but this is an optional feature and has to be activated explicit:

```python
from ezdxf import options
options.use_matplotlib_font_support = True
```

This is a global option for the current running Interpreter and it is active until deactivated:

```python
options.use_matplotlib_font_support = False
```

**Warning:** This feature requires a working Matplotlib installation else an ImportError exception will be raised sooner or later. This feature also depends on the drawing add-on, which is installed by default. Using the Matplotlib font support adds runtime overhead at the first usage of any of the text related primitives.

See also:

Global option to set the font caching directory: ezdxf.options.font_cache_directory

Install Matplotlib from command line:

```bash
pip3 install matplotlib
```

The Matplotlib font support will improve the results for TEXT, ATTRIB and ATTDEF. The MTEXT entity has many advanced features which would require a full “Rich Text Format” rendering and that is far beyond the goals and capabilities of this library, therefore the boundary box for MTEXT will never be as accurate as in a dedicated CAD application.
Flatten Complex DXF Entities

```python
recursive_decompose(entities: Iterable[DXFEntity]) → Iterable[DXFEntity]
```

Recursive decomposition of the given DXF entity collection into a flat DXF entity stream. All block references (INSERT) and entities which provide a `virtual_entities()` method will be disassembled into simple DXF sub-entities, therefore the returned entity stream does not contain any INSERT entity.

Point entities will **not** be disassembled into DXF sub-entities, as defined by the current point style `$PDMODE$`.

These entity types include sub-entities and will be decomposed into simple DXF entities:

- INSERT
- DIMENSION
- LEADER
- MLEADER
- MLINE

Decomposition of XREF, UNDERLAY and ACAD_TABLE entities is not supported.

Entity Deconstruction

These functions disassemble DXF entities into simple geometric objects like meshes, paths or vertices. The `Primitive` is a simplified intermediate class to use a common interface on various DXF entities.

```python
make_primitive(entity: DXFEntity, max_flattening_distance=None) → Primitive
```

Factory to create path/mesh primitives. The `max_flattening_distance` defines the max distance between the approximation line and the original curve. Use `max_flattening_distance` to override the default value.

Returns an empty primitive for unsupported entities. The `empty state of a primitive can be checked by the property `is_empty`. The `path` and the `mesh` attributes of an empty primitive are `None` and the `vertices()` method yields no vertices.

Returns an empty primitive for the ` Hatch` entity, see docs of the `disassemble` module. Use this to create multiple primitives from the HATCH boundary paths:

```python
primitives = list(to_primitives([hatch_entity]))
```

```python
to_primitives(entities: Iterable[DXFEntity], max_flattening_distance: float = None) → Iterable[Primitive]
```

Yields all DXF entities as path or mesh primitives. Yields unsupported entities as empty primitives, see `make_primitive()`.

**Parameters**

- `entities` – iterable of DXF entities
- `max_flattening_distance` – override the default value

```python
to_meshes(primitives: Iterable[Primitive]) → Iterable[MeshBuilder]
```

Yields all `MeshBuilder` objects from the given `primitives`. Ignores primitives without a defined mesh.

```python
to_paths(primitives: Iterable[Primitive]) → Iterable[Path]
```

Yields all `Path` objects from the given `primitives`. Ignores primitives without a defined path.
ezdxf.disassemble.to_vertices (primitives: Iterable[Primitive]) → Iterable[Vec3]
Yields all vertices from the given primitives. Paths will be flattened to create the associated vertices. See also to_control_vertices() to collect only the control vertices from the paths without flattening.

ezdxf.disassemble.to_control_vertices (primitives: Iterable[Primitive]) → Iterable[Vec3]
Yields all path control vertices and all mesh vertices from the given primitives. Like to_vertices(), but without flattening.

class ezdxf.disassemble.Primitive
Interface class for path/mesh primitives.

def entity
Reference to the source DXF entity of this primitive.

def max_flattening_distance
The max_flattening_distance attribute defines the max distance in drawing units between the approximation line and the original curve. Set the value by direct attribute access. (float) default = 0.01

def vertices () → Iterable[Vec3]
Yields all vertices of the path/mesh representation as Vec3 objects.

Bounding Box

New in version 0.16. The ezdxf.bbox module provide tools to calculate bounding boxes for many DXF entities, but not for all. The bounding box calculation is based on the ezdxf.disassemble module and therefore has the same limitation.

Warning: If accurate boundary boxes for text entities are important for you, read this first: Text Boundary Calculation. TL;DR: Boundary boxes for text entities are not accurate!

Unsupported DXF entities:
- All ACIS based types like BODY, 3DSOLID or REGION
- External references (XREF) and UNDERLAY object
- RAY and XRAY, extend into infinite
- ACAD_TABLE, no basic support - only preserved by ezdxf

Unsupported entities are silently ignored, filtering of these DXF types is not necessary.

The base type for bounding boxes is the BoundingBox class from the module ezdxf.math.

The entities iterable as input can be the whole modelspace, an entity query or any iterable container of DXF entities.

The Calculation of bounding boxes of curves is done by flattening the curve by a default flattening distance of 0.01. Set argument flatten to 0 to speedup the bounding box calculation by accepting less precision for curved objects by using only the control vertices.

The optional caching object Cache has to be instantiated by the user, this is only useful if the same entities will be processed multiple times.

Example usage with caching:

```python
from ezdxf import bbox
msp = doc.modelspace()
```
cache = bbox.Cache()
# get overall bounding box
first_bbox = bbox.extents(msp, cache=cache)
# bounding box of all LINE entities
second_bbox = bbox.extend(msp.query("LINE"), cache=cache)

## Functions

**ezdxf.bbox.extents** *(entities: Iterable[DXFEntity], *, flatten=0.01, cache: Cache=None) → BoundingBox*

Returns a single bounding box for all given entities.

Calculate bounding boxes from flattened curves, if argument *flatten* is not 0 (max flattening distance), else from control points.

**ezdxf.bbox.multi_flat** *(entities: Iterable[DXFEntity], *, flatten=0.01, cache: Cache=None) → Iterable[BoundingBox]*

Yields a bounding box for each of the given entities.

Calculate bounding boxes from flattened curves, if argument *flatten* is not 0 (max flattening distance), else from control points.

**ezdxf.bbox.multi_recursive** *(entities: Iterable[DXFEntity], *, flatten=0.01, cache: Cache=None) → Iterable[BoundingBox]*

Yields all bounding boxes for the given entities or all bounding boxes for their sub entities. If an entity (INSERT) has sub entities, only the bounding boxes of these sub entities will be yielded, not the bounding box of entity (INSERT) itself.

Calculate bounding boxes from flattened curves, if argument *flatten* is not 0 (max flattening distance), else from control points.

### Caching Strategies

Because *ezdxf* is not a CAD application, *ezdxf* does not manage data structures which are optimized for a usage by a CAD kernel. This means that the content of complex entities like block references or leaders has to be created on demand by DXF primitives on the fly. These temporarily created entities are called virtual entities and have no handle and are not stored in the entities database.

All this is required to calculate the bounding box of complex entities, and it is therefore a very time consuming task. By using a *Cache* object it is possible to speedup this calculations, but this is not a magically feature which requires an understanding of what is happening under the hood to achieve any performance gains.

For a single bounding box calculation, without any reuse of entities it makes no sense of using a *Cache* object, e.g. calculation of the modelspace extents:

```python
from pathlib import Path
import ezdxf
from ezdxf import bbox

CADKitSamples = Path(ezdxf.EZDXF_TEST_FILES) / 'CADKitSamples'

doc = ezdxf.readfile(CADKitSamples / 'A_000217.dxf')
cache = bbox.Cache()

ext = bbox.extents(doc.modelspace(), cache)
```
1226 cached objects and not a single cache hit:

```
Cache(n=1226, hits=0, misses=3273)
```

The result for using UUIDs to cache virtual entities is not better:

```
Cache(n=2206, hits=0, misses=3273)
```

Same count of hits and misses, but now the cache also references ~1000 virtual entities, which block your memory until the cache is deleted, luckily this is a small DXF file (~838 kB).

Bounding box calculations for multiple entity queries, which have overlapping entity results, using a Cache object may speedup the calculation:

```
doc = ezdxf.readfile(CADKitSamples / 'A_000217.dxf')
msp = doc.modelspace()
cache = bbox.Cache(uuid=False)
ext = bbox.extents(msp, cache)
print(cache)

# process modelspace again
ext = bbox.extents(msp, cache)
print(cache)
```

Processing the same data again leads some hits:

```
1st run: Cache(n=1226, hits=0, misses=3273)
2nd run: Cache(n=1226, hits=1224, misses=3309)
```

Using uuid=True leads not to more hits, but more cache entries:

```
1st run: Cache(n=2206, hits=0, misses=3273)
2nd run: Cache(n=2206, hits=1224, misses=3309)
```

Creating stable virtual entities by disassembling the entities at first leads to more hits:

```
from ezdxf import disassemble
disassemble.recursive_decompose(msp))
cache = bbox.Cache(uuid=False)
ext = bbox.extents(entities, cache)
print(cache)

bbox.extents(entities, cache)
print(cache)
```

First without UUID for stable virtual entities:

```
1st run: Cache(n=1037, hits=0, misses=4074)
2nd run: Cache(n=1037, hits=1037, misses=6078)
```

Using UUID for stable virtual entities leads to more hits:
But caching virtual entities needs also more memory.

In conclusion: Using a cache is only useful, if you often process nearly the same data; only then can a performance gain be expected.

**Cache Class**

```python
class ezdxf.bbox.Cache(uuid=False)
    Caching object for `ezdxf.math.BoundingBox` objects.

    Parameters
    ----------
    uuid : bool, optional
        use UUIDs for virtual entities

    hits

    misses

    invalidate(entities: Iterable[DXFEntity]) → None
        Invalidate cache entries for the given DXF entities.

        If entities are changed by the user, it is possible to invalidate individual entities. Use with care - discarding the whole cache is the safer workflow.

        Ignores entities which are not stored in cache.
```

### 6.8.6 Global Options

**Options Module**

Global options stored in `ezdxf.options`

- `ezdxf.options.default_text_style`  Default text styles, default value is OpenSans.
- `ezdxf.options.default_dimension_text_style`  Default text style for Dimensions, default value is OpenSansCondensed-Light.
- `ezdxf.options.use_matplotlib`  Activate/deactivate Matplotlib support (e.g. for testing) if Matplotlib is installed, else `use_matplotlib` is always False.
- `ezdxf.options.font_cache_directory`  Set path to an external font cache directory: e.g. "~/.ezdxf"

By default the bundled font cache will be loaded.

This example shows, how to create an external font cache in "~/.ezdxf". This has to be done only once after the `ezdxf` installation or to add new installed fonts to the cache. This requires Matplotlib:

```python
from ezdxf.tools import fonts

fonts.build_system_font_cache(path="~/.ezdxf")
```

How to use this cached font data in your script:
from ezdxf import options, fonts

option.font_cache_directory = "~/.ezdxf"
# Default cache is loaded automatically, if auto load is not disabled:
fonts.load(reload=True)

Maybe it is better to set an environment variable before ezdxf is loaded:

C:\> set EZDXF_FONT_CACHE_DIRECTORY=~/.ezdxf

ezdxf.options.filter_invalid_xdata_group_codes
    Filter invalid XDATA group codes, default value is False.

ezdxf.options.log_unprocessed_tags
    Log unprocessed DXF tags for debugging, default value is True.

ezdxf.options.write_fixed_meta_data_for_testing
    Enable this option to always create same meta data for testing scenarios, e.g. to use a diff like tool to compare DXF documents, default is False.

ezdxf.options.load_proxy_graphics
    Load proxy graphics if True, default is False.

Important: If you change any environment variable, you have to restart the Python interpreter!

EZDXF_DISABLE_C_EXT Set environment variable EZDXF_DISABLE_C_EXT to 1 or True to disable the usage of C extensions implemented by Cython. Disabling the C-extensions can only be done on interpreter startup, before the first import of ezdxf.

EZDXF_AUTO_LOAD_FONTS Set EZDXF_AUTO_LOAD_FONTS to 0 or False to deactivate font cache loading at startup, if this slows down the interpreter startup too much and font measuring is not important to you. The font cache can always be loaded manually by calling ezdxf.fonts.load()

EZDXF_FONT_CACHE_DIRECTORY Set path to an external font cache directory, see font_cache_directory for more information

EZDXF_TEST_FILES Path to the ezdxf test files required by some tests, for instance the CADKit sample files should be located in the “EZDXF_TEST_FILES/CADKitSamples” folder.

EZDXF_PRESERVE_PROXY_GRAPHICS Enable proxy graphic load/store support if 1 or True, default value is True.

EZDXF_LOG_UNPROCESSED_TAGS Log unprocessed DXF tags for debugging, default value is True.
EZDXF_FILTER_INVALID_XDATA_GROUP_CODES Filter invalid XDATA group codes, default value is False.

6.8.7 Miscellaneous

Zoom Layouts

New in version 0.16.

These functions mimic the ZOOM commands in CAD applications.

Zooming means resetting the current viewport limits to new values. The coordinates for the functions `center()` and `window()` are drawing units for the model space and paper space units for paper space layouts. The modelspace units in `Drawing.units` are ignored.

The extents detection for the functions `entities()` and `extents()` is done by the `ezdxf.bbox` module. Read the associated documentation to understand the limitations of the `ezdxf.bbox` module. Tl;dr The extents detection is slow and not accurate.

Because the ZOOM operations in CAD applications are not that precise, then zoom functions of this module uses the fast bounding box calculation mode of the `bbox` module, which means the argument `flatten` is always `False` for `extents()` function calls.

The region displayed by CAD applications also depends on the aspect ratio of the application window, which is not available to `ezdxf`, therefore the viewport size is just an educated guess of an aspect ratio of 2:1 (16:9 minus top toolbars and the bottom status bar).

**Warning:** All zoom functions replace the current viewport configuration by a single window configuration.

Example to reset the main CAD viewport of the model space to the extents of its entities:

```python
import ezdxf
from ezdxf import zoom

doc = ezdxf.new()
msp = doc.modelspace()
... # add your DXF entities

zoom.extents(msp)
doc.saveas("your.dxf")
```

- `ezdxf.zoom.center(layout: Layout, point: Vertex, height: float)`
  Resets the active viewport center of `layout` to the given `point`, argument `size` defines the width and height of the viewport. Replaces the current viewport configuration by a single window configuration.

- `ezdxf.zoom.objects(layout: Layout, entities: Iterable[DXFEntity], factor: float=1)`
  Resets the active viewport limits of `layout` to the extents of the given `entities`. Only entities in the given `layout` are taken into account. The argument `factor` scales the viewport limits. Replaces the current viewport configuration by a single window configuration.

- `ezdxf.zoom.extents(layout: Layout, factor: float=1)`
  Resets the active viewport limits of `layout` to the extents of all entities in this `layout`. The argument `factor` scales the viewport limits. Replaces the current viewport configuration by a single window configuration.

- `ezdxf.zoom.window(layout: Layout, p1: Vertex, p2: Vertex)`
  Resets the active viewport limits of `layout` to the lower left corner `p1` and the upper right corner `p2`. Replaces the current viewport configuration by a single window configuration.
Load DXF Comments

ezdxf.comments.from_stream(stream: TextIO, codes: Set[int] = None) → Iterable[DXFTag]
Yields comment tags from text stream as DXFTag objects.

Parameters

• stream – input text stream
• codes – set of group codes to yield additional DXF tags e.g. {5, 0} to also yield handle and structure tags

ezdxf.comments.from_file(filename: str, codes: Set[int] = None) → Iterable[DXFTag]
Yields comment tags from file filename as DXFTag objects.

Parameters

• filename – filename as string
• codes – yields also additional tags with specified group codes e.g. {5, 0} to also yield handle and structure tags

Tools

DXF Unicode Decoder

The DXF format uses a special form of unicode encoding: “\U+xxxx”. 
To avoid a speed penalty such encoded characters are not decoded automatically by the regular loading function: ezdxf.readfile, only the recover module does the decoding automatically, because this loading mode is already slow.

This kind of encoding is most likely used only in older DXF versions, because since DXF R2007 the whole DXF file is encoded in utf8 and a special unicode encoding is not necessary.

The ezdxf.has_dxf_unicode() and ezdxf.decode_dxf_unicode() are new support functions to decode unicode characters “\U+xxxx” manually.

New in version 0.14.

ezdxf.has_dxf_unicode(s: str) → bool
Detect if string s contains encoded DXF unicode characters “\U+xxxx”.

ezdxf.decode_dxf_unicode(s: str) → str
Decode DXF unicode characters “\U+xxxx” in string s.

Tools

Some handy tool functions used internally by ezdxf.

ezdxf.int2rgb(value: int) → Tuple[int, int, int]
Split RGB integer value into (r, g, b) tuple.

ezdxf.rgb2int(rgb: Tuple[int, int, int]) → int
Combined integer value from (r, g, b) tuple.

ezdxf.float2transparency(value: float) → int
Returns DXF transparency value as integer in the range from 0 to 255, where 0 is 100% transparent and 255 is opaque.
Parameters **value** – transparency value as float in the range from 0 to 1, where 0 is opaque and 1 is 100% transparent.

```python
import ezdxf

ezdxf.transparency2float(value: int) -> float
```

Returns transparency value as float from 0 to 1, 0 for no transparency (opaque) and 1 for 100% transparency.

Parameters **value** – DXF integer transparency value, 0 for 100% transparency and 255 for opaque

```python
import ezdxf

ezdxf.colors.aci2rgb(index: int) -> Tuple[int, int, int]
```

Convert AutoCAD Color Index (ACI) into \((r, g, b)\) tuple, based on default AutoCAD colors.

```python
import ezdxf

ezdxf.colors.luminance(color: Tuple[int, int, int]) -> float
```

Returns perceived luminance for a RGB color in the range \([0.0, 1.0]\) from dark to light.

```python
import ezdxf

ezdxf.tools.juliandate(date: datetime.datetime) -> float
```

```python
import ezdxf

ezdxf.tools.calendardate(juliandate: float) -> datetime.datetime
```

```python
import ezdxf

ezdxf.tools.set_flag_state(flags: int, flag: int, state: bool = True) -> int
```

Set/clear binary flag in data flags.

Parameters

- **flags** – data value
- **flag** – flag to set/clear
- **state** – True for setting, False for clearing

```python
import ezdxf

ezdxf.tools.guid() -> str
```

Returns a general unique ID, based on `uuid.uuid4()`.

This function creates a GUID for the header variables `$VERSIONGUID` and `$FINGERPRINTGUID`, which matches the AutoCAD pattern `{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}`.

```python
import ezdxf

ezdxf.tools.bytes_to_hexstr(data: bytes) -> str
```

Returns data bytes as plain hex string.

```python
import ezdxf

ezdxf.tools.suppress_zeros(s: str, leading: bool = False, trailing: bool = True)
```

Suppress trailing and/or leading 0 of string `s`.

Parameters

- **s** – data string
- **leading** – suppress leading 0
- **trailing** – suppress trailing 0

```python
import ezdxf

ezdxf.tools.normalize_text_angle(angle: float, fix_upside_down=False) -> float
```

Normalizes text angle to the range from 0 to 360 degrees and fixes upside down text angles.

Parameters

- **angle** – text angle in degrees
- **fix_upside_down** – rotate upside down text angle about 180 degree

### SAT Format “Encryption”

```python
import ezdxf

ezdxf.tools.crypt.encode(text_lines: Iterable[str]) -> Iterable[str]
```

Encode the Standard ACIS Text (SAT) format by AutoCAD “encryption” algorithm.

```python
import ezdxf

ezdxf.tools.crypt.decode(text_lines: Iterable[str]) -> Iterable[str]
```

Decode the Standard ACIS Text (SAT) format “encrypted” by AutoCAD.
Reorder

Tools to reorder DXF entities by handle or a special sort handle mapping.

Such reorder mappings are stored only in layouts as Modelspace, Paperspace or BlockLayout, and can be retrieved by the method get_redraw_order().

Each entry in the handle mapping replaces the actual entity handle, where the “0” handle has a special meaning, this handle always shows up at last in ascending ordering.

```python
ezdxf.reorder.ascending(entities: Iterable[DXFGraphic], mapping: Union[Dict[KT, VT], Iterable[Tuple[str, str]]] = None) -> Iterable[DXFGraphic]
```

Yields entities in ascending handle order.

The sort handle doesn’t have to be the entity handle, every entity handle in mapping will be replaced by the given sort handle, mapping is an iterable of 2-tuples (entity_handle, sort_handle) or a dict (entity_handle, sort_handle). Entities with equal sort handles show up in source entities order.

Parameters

- `entities` – iterable of DXFGraphic objects
- `mapping` – iterable of 2-tuples (entity_handle, sort_handle) or a handle mapping as dict.

```python
ezdxf.reorder.descending(entities: Iterable[DXFGraphic], mapping: Union[Dict[KT, VT], Iterable[Tuple[str, str]]] = None) -> Iterable[DXFGraphic]
```

Yields entities in descending handle order.

The sort handle doesn’t have to be the entity handle, every entity handle in mapping will be replaced by the given sort handle, mapping is an iterable of 2-tuples (entity_handle, sort_handle) or a dict (entity_handle, sort_handle). Entities with equal sort handles show up in reversed source entities order.

Parameters

- `entities` – iterable of DXFGraphic objects
- `mapping` – iterable of 2-tuples (entity_handle, sort_handle) or a handle mapping as dict.

6.9 Rendering

The ezdxf.render subpackage provides helpful utilities to create complex forms.

- create complex meshes as Mesh entity.
- render complex curves like bezier curves, euler spirals or splines as Polyline entity
- vertex generators for simple and complex forms like circle, ellipse or euler spiral

Content

6.9.1 Spline

```python
class ezdxf.render.Spline(points: Iterable[Vertex] = None, segments: int = 100)
```

This class can be used to render B-splines into DXF R12 files as approximated Polyline entities. The advantage of this class over the R12Spline class is, that this is a real 3D curve, which means that the B-spline vertices do have to be located in a flat plane, and no UCS class is needed to place the curve in 3D space.

See also:

The newer BSpline class provides the advanced vertex interpolation method flattening().
ezdxf Documentation, Release 0.16.2

```python
__init__(points: Iterable[Vertex] = None, segments: int = 100)

Parameters

• points – spline definition points
• segments – count of line segments for approximation, vertex count is segments + 1

subdivide(segments: int = 4) → None

Calculate overall segment count, where segments is the sub-segment count, segments = 4, means 4 line segments between two definition points e.g. 4 definition points and 4 segments = 12 overall segments, useful for fit point rendering.

Parameters

• segments – sub-segments count between two definition points

render_as_fit_points(layout: BaseLayout, degree: int = 3, method: str = 'chord', dxfattribs: dict = None) → None

Render a B-spline as 2D/3D Polyline, where the definition points are fit points.

• 2D spline vertices uses: add_polyline2d()
• 3D spline vertices uses: add_polyline3d()

Parameters

• layout – BaseLayout object
• degree – degree of B-spline (order = degree + 1)
• method – “uniform”, “distance”/”chord”, “centripetal”/”sqrt_chord” or “arc” calculation method for parameter t
• dxfattribs – DXF attributes for Polyline

render_open_bspline(layout: BaseLayout, degree: int = 3, dxfattribs: dict = None) → None

Render an open uniform B-spline as 3D Polyline. Definition points are control points.

Parameters

• layout – BaseLayout object
• degree – degree of B-spline (order = degree + 1)
• dxfattribs – DXF attributes for Polyline

render_uniform_bspline(layout: BaseLayout, degree: int = 3, dxfattribs: dict = None) → None

Render a uniform B-spline as 3D Polyline. Definition points are control points.

Parameters

• layout – BaseLayout object
• degree – degree of B-spline (order = degree + 1)
• dxfattribs – DXF attributes for Polyline

render_closed_bspline(layout: BaseLayout, degree: int = 3, dxfattribs: dict = None) → None

Render a closed uniform B-spline as 3D Polyline. Definition points are control points.

Parameters

• layout – BaseLayout object
• degree – degree of B-spline (order = degree + 1)
• dxfattribs – DXF attributes for Polyline
```
**render_open_rbspline** *(layout: BaseLayout, weights: Iterable[float], degree: int = 3, dxfattribs: dict = None) → None*

Render a rational open uniform BSpline as 3D *Polyline*. Definition points are control points.

**Parameters**

- `layout` – *BaseLayout* object
- `weights` – list of weights, requires a weight value (float) for each definition point.
- `degree` – degree of B-spline (order = degree + 1)
- `dxfattribs` – DXF attributes for *Polyline*

**render_uniform_rbspline** *(layout: BaseLayout, weights: Iterable[float], degree: int = 3, dxfattribs: dict = None) → None*

Render a rational uniform B-spline as 3D *Polyline*. Definition points are control points.

**Parameters**

- `layout` – *BaseLayout* object
- `weights` – list of weights, requires a weight value (float) for each definition point.
- `degree` – degree of B-spline (order = degree + 1)
- `dxfattribs` – DXF attributes for *Polyline*

**render_closed_rbspline** *(layout: BaseLayout, weights: Iterable[float], degree: int = 3, dxfattribs: dict = None) → None*

Render a rational B-spline as 3D *Polyline*. Definition points are control points.

**Parameters**

- `layout` – *BaseLayout* object
- `weights` – list of weights, requires a weight value (float) for each definition point.
- `degree` – degree of B-spline (order = degree + 1)
- `dxfattribs` – DXF attributes for *Polyline*

### 6.9.2 R12Spline

**class** `ezdxf.render.R12Spline` *(control_points: Iterable[Vertex], degree: int = 2, closed: bool = True)*

DXF R12 supports 2D B-splines, but Autodesk do not document the usage in the DXF Reference. The base entity for splines in DXF R12 is the POLYLINE entity. The spline itself is always in a plane, but as any 2D entity, the spline can be transformed into the 3D object by elevation and extrusion (*OCS*, *UCS*).

This way it was possible to store the spline parameters in the DXF R12 file, to allow CAD applications to modify the spline parameters and rerender the B-spline afterward again as polyline approximation. Therefore the result is not better than an approximation by the *Spline* class, it is also just a POLYLINE entity, but maybe someone need exact this tool in the future.

**__init__** *(control_points: Iterable[Vertex], degree: int = 2, closed: bool = True)*

**Parameters**

- `control_points` – B-spline control frame vertices
- `degree` – degree of B-spline, only 2 and 3 is supported
- `closed` – True for closed curve
render (layout: BaseLayout, segments: int = 40, ucs: UCS = None, dxfattribs: dict = None) → Polyline
Renders the B-spline into layout as 2D Polyline entity. Use an UCS to place the 2D spline in the 3D space, see approximate() for more information.

Parameters
- **layout** – BaseLayout object
- **segments** – count of line segments for approximation, vertex count is segments + 1
- **ucs** – UCS definition, control points in ucs coordinates.
- **dxfattribs** – DXF attributes for Polyline

approximate (segments: int = 40, ucs: UCS = None) → List[Vertex]
Approximate the B-spline by a polyline with segments line segments. If ucs is not None, ucs defines an UCS, to transformed the curve into OCS. The control points are placed xy-plane of the UCS, don’t use z-axis coordinates, if so make sure all control points are in a plane parallel to the OCS base plane (UCS xy-plane), else the result is unpredictable and depends on the CAD application used to open the DXF file - it may crash.

Parameters
- **segments** – count of line segments for approximation, vertex count is segments + 1
- **ucs** – UCS definition, control points in ucs coordinates

Returns list of vertices in OCS as Vec3 objects

6.9.3 Bezier

class ezdxf.render.Bezier
Render a bezier curve as 2D/3D Polyline.

The Bezier class is implemented with multiple segments, each segment is an optimized 4 point bezier curve, the 4 control points of the curve are: the start point (1) and the end point (4), point (2) is start point + start vector and point (3) is end point + end vector. Each segment has its own approximation count.

See also:
The new ezdxf.path package provides many advanced construction tools based on the Path class.

start (point: Vertex, tangent: Vertex) → None
Set start point and start tangent.

Parameters
- **point** – start point
- **tangent** – start tangent as vector, example: (5, 0, 0) means a horizontal tangent with a length of 5 drawing units

append (point: Vertex, tangent1: Vertex, tangent2: Vertex = None, segments: int = 20)
Append a control point with two control tangents.

Parameters
- **point** – control point
- **tangent1** – first tangent as vector “left” of the control point
- **tangent2** – second tangent as vector “right” of the control point, if omitted tangent2 = -tangent1
• `segments` – count of line segments for the polyline approximation, count of line segments from the previous control point to the appended control point.

```python
render (layout: BaseLayout, force3d: bool = False, dxfattribs: dict = None) \rightarrow None
```

Render bezier curve as 2D/3D Polyline.

Parameters

- `layout` – `BaseLayout` object
- `force3d` – force 3D polyline rendering
- `dxfattribs` – DXF attributes for Polyline

### 6.9.4 EulerSpiral

class ezdxf.render.EulerSpiral (curvature: float = 1)

Render an euler spiral as a 3D Polyline or a Spline entity.

This is a parametric curve, which always starts at the origin (0, 0).

```python
__init__ (curvature: float = 1)
```

Parameters `curvature` – Radius of curvature

```python
render_polyline (layout: BaseLayout, length: float = 1, segments: int = 100, matrix: Matrix44 = None, dxfattribs: dict = None)
```

Render curve as Polyline.

Parameters

- `layout` – `BaseLayout` object
- `length` – length measured along the spiral curve from its initial position
- `segments` – count of line segments to use, vertex count is `segments` + 1
- `matrix` – transformation matrix as `Matrix44`
- `dxfattribs` – DXF attributes for Polyline

Returns Polyline

```python
render_spline (layout: BaseLayout, length: float = 1, fit_points: int = 10, degree: int = 3, matrix: Matrix44 = None, dxfattribs: dict = None)
```

Render curve as Spline.

Parameters

- `layout` – `BaseLayout` object
- `length` – length measured along the spiral curve from its initial position
- `fit_points` – count of spline fit points to use
- `degree` – degree of B-spline
- `matrix` – transformation matrix as `Matrix44`
- `dxfattribs` – DXF attributes for Spline

Returns Spline
6.9.5 Random Paths

Random path generators for testing purpose.

```python
def random_2d_path(steps=100, max_step_size=1, max_heading=\pi/2, retarget=20)
    """Returns a random 2D path as iterable of Vec2 objects."
    """
    yield Vec2
```

Parameters

- `steps` – count of vertices to generate
- `max_step_size` – max step size
- `max_heading` – limit heading angle change per step to $\pm \max_heading/2 \text{ in radians}$
- `retarget` – specifies steps before changing global walking target

```python
def random_3d_path(steps=100, max_step_size=1, max_heading=\pi/2, max_pitch=\pi/8, retarget=20)
    """Returns a random 3D path as iterable of Vec3 objects."
    """
    yield Vec3
```

Parameters

- `steps` – count of vertices to generate
- `max_step_size` – max step size
- `max_heading` – limit heading angle change per step to $\pm \max_heading/2$, rotation about the z-axis in radians
- `max_pitch` – limit pitch angle change per step to $\pm \max_pitch/2$, rotation about the x-axis in radians
- `retarget` – specifies steps before changing global walking target

6.9.6 Forms

This module provides functions to create 2D and 3D forms as vertices or mesh objects.

2D Forms

- `circle()`
- `square()`
- `box()`
- `ellipse()`
- `euler_spiral()`
- `ngon()`
- `star()`
- `gear()`

3D Forms

- `cube()`
- `cylinder()`
- `cylinder_2p()`
- `cone()`
• cone_2p()
• sphere()

3D Form Builder
• extrude()
• from_profiles_linear()
• from_profiles_spline()
• rotation_form()

2D Forms

Basic 2D shapes as iterable of Vec3.

```python
ezdxf.render.forms.circle(count: int, radius: float = 1, elevation: float = 0, close: bool = False) -> Iterable[Vec3]
```

Create polygon vertices for a circle with the given radius and approximated by count vertices, elevation is the z-axis for all vertices.

**Parameters**
- `count` – count of polygon vertices
- `radius` – circle radius
- `elevation` – z-axis for all vertices
- `close` – yields first vertex also as last vertex if True.

**Returns** vertices in counter clockwise orientation as Vec3 objects

```python
ezdxf.render.forms.square(size: float = 1.) -> Tuple[Vec3, Vec3, Vec3, Vec3]
```

Returns 4 vertices for a square with a side length of the given size, lower left corner is (0, 0), upper right corner is (size, size).

```python
ezdxf.render.forms.box(sx: float = 1., sy: float = 1.) -> Tuple[Vec3, Vec3, Vec3, Vec3]
```

Returns 4 vertices for a box with a width of sx by and a height of sy, lower left corner is (0, 0), upper right corner is (sx, sy).

```python
ezdxf.render.forms.ellipse(count: int, rx: float = 1, ry: float = 1, start_param: float = 0, end_param: float = 2 * pi, elevation: float = 0) -> Iterable[Vec3]
```

Create polygon vertices for an ellipse with given rx as x-axis radius and ry as y-axis radius approximated by count vertices, elevation is the z-axis for all vertices. The ellipse goes from start_param to end_param in counter clockwise orientation.

**Parameters**
- `count` – count of polygon vertices
- `rx` – ellipse x-axis radius
- `ry` – ellipse y-axis radius
- `start_param` – start of ellipse in range [0, 2π]
- `end_param` – end of ellipse in range [0, 2π]
- `elevation` – z-axis for all vertices

**Returns** vertices in counter clockwise orientation as Vec3 objects

6.9. Rendering
ezdxf.render.forms.euler_spiral(count: int, length: float = 1, curvature: float = 1, elevation: float = 0) → Iterable[Vec3]

Create polygon vertices for an euler spiral of a given length and radius of curvature. This is a parametric curve, which always starts at the origin (0, 0).

Parameters

- **count** – count of polygon vertices
- **length** – length of curve in drawing units
- **curvature** – radius of curvature
- **elevation** – z-axis for all vertices

Returns vertices as Vec3 objects

ezdxf.render.forms.ngon(count: int, length: float = None, radius: float = None, rotation: float = 0., elevation: float = 0., close: bool = False) → Iterable[Vec3]

Returns the corner vertices of a regular polygon. The polygon size is determined by the edge length or the circum radius argument. If both are given length has the higher priority.

Parameters

- **count** – count of polygon corners >= 3
- **length** – length of polygon side
- **radius** – circum radius
- **rotation** – rotation angle in radians
- **elevation** – z-axis for all vertices
- **close** – yields first vertex also as last vertex if True.

Returns vertices as Vec3 objects

ezdxf.render.forms.star(count: int, r1: float, r2: float, rotation: float = 0., elevation: float = 0., close: bool = False) → Iterable[Vec3]

Returns the corner vertices for a star shape.

The shape has count spikes, r1 defines the radius of the “outer” vertices and r2 defines the radius of the “inner” vertices, but this does not mean that r1 has to be greater than r2.

Parameters

- **count** – spike count >= 3
- **r1** – radius 1
- **r2** – radius 2
- **rotation** – rotation angle in radians
- **elevation** – z-axis for all vertices
- **close** – yields first vertex also as last vertex if True.

Returns vertices as Vec3 objects

ezdxf.render.forms.gear(count: int, top_width: float, bottom_width: float, height: float, outside_radius: float, elevation: float = 0, close: bool = False) → Iterable[Vec3]

Returns the corner vertices of a gear shape (cogwheel).
Warning: This function does not create correct gears for mechanical engineering!

Parameters

- `count` – teeth count >= 3
- `top_width` – teeth width at outside radius
- `bottom_width` – teeth width at base radius
- `height` – teeth height; base radius = outside radius - height
- `outside_radius` – outside radius
- `elevation` – z-axis for all vertices
- `close` – yields first vertex also as last vertex if True.

Returns vertices in counter clockwise orientation as `Vec3` objects

3D Forms

Create 3D forms as `MeshTransformer` objects.

```python
ezdxfrfrend:forms.cube(center: bool = True) → MeshTransformer

Create a cube as `MeshTransformer` object.
```

Parameters `center` – ‘mass’ center of cube, (0, 0, 0) if True, else first corner at (0, 0, 0)

Returns: `MeshTransformer`

```python
ezdxfrfrend:forms.cylinder(count: int, radius: float = 1., top_radius: float = None, top_center: Vertex = (0, 0, 1), caps=True, ngons=True) → MeshTransformer

Create a cylinder as `MeshTransformer` object, the base center is fixed in the origin (0, 0, 0).
```

Parameters

- `count` – profiles edge count
- `radius` – radius for bottom profile
- `top_radius` – radius for top profile, if None top_radius == radius
- `top_center` – location vector for the center of the top profile
- `caps` – close hull with bottom cap and top cap (as N-gons)
- `ngons` – use ngons for caps if True else subdivide caps into triangles

Returns: `MeshTransformer`

```python
ezdxfrfrend:forms.cylinder_2p(count: int = 16, radius: float = 1., base_center=(0, 0, 0), top_center=(0, 0, 1)) → MeshTransformer

Create a cylinder as `MeshTransformer` object from two points, base_center is the center of the base circle and, top_center the center of the top circle.
```

Parameters

- `count` – profiles edge count
- `radius` – radius for bottom profile
- `base_center` – center of base circle
• **top_center** – center of top circle

Returns: *MeshTransformer*

ezdxf.render.forms.cone*(count: int, radius: float, apex: Vertex = (0, 0, 1), caps=True, ngons=True)* → MeshTransformer

Create a cone as *MeshTransformer* object, the base center is fixed in the origin (0, 0, 0).

**Parameters**

• **count** – edge count of basis_vector
• **radius** – radius of basis_vector
• **apex** – tip of the cone
• **caps** – add a bottom face if True
• **ngons** – use ngons for caps if True else subdivide caps into triangles

Returns: *MeshTransformer*

ezdxf.render.forms.cone_2p*(count: int, radius: float, apex: Vertex = (0, 0, 1))* → MeshTransformer

Create a cone as *MeshTransformer* object from two points, base_center is the center of the base circle and apex as the tip of the cone.

**Parameters**

• **count** – edge count of basis_vector
• **radius** – radius of basis_vector
• **base_center** – center point of base circle
• **apex** – tip of the cone

Returns: *MeshTransformer*

ezdxf.render.forms.sphere*(count: int = 16, stacks: int = 8, radius: float = 1, quads=True)* → MeshTransformer

Create a sphere as *MeshTransformer* object, center is fixed at origin (0, 0, 0).

**Parameters**

• **count** – longitudinal slices
• **stacks** – latitude slices
• **radius** – radius of sphere
• **quads** – use quads for body faces if True else triangles

Returns: *MeshTransformer*

---

**3D Form Builder**

ezdxf.render.forms.extrude*(profile: Iterable[Vertex], path: Iterable[Vertex], close=True)* → MeshTransformer

Extrude a profile polygon along a path polyline, vertices of profile should be in counter clockwise order.

**Parameters**

• **profile** – sweeping profile as list of (x, y, z) tuples in counter clockwise order
• **path** – extrusion path as list of (x, y, z) tuples
• **close** – close profile polygon if True
Returns: MeshTransformer
ezdxf.render.forms.from_profiles_linear(profiles: Iterable[Iterable[Vertex]], close=True, caps=False, ngons=True) → MeshTransformer
Create MESH entity by linear connected profiles.

Parameters
• profiles – list of profiles
• close – close profile polygon if True
• caps – close hull with bottom cap and top cap
• ngons – use ngons for caps if True else subdivide caps into triangles

Returns: MeshTransformer
ezdxf.render.forms.from_profiles_spline(profiles: Iterable[Iterable[Vertex]], subdivide: int = 4, close=True, caps=False, ngons=True) → MeshTransformer
Create MESH entity by spline interpolation between given profiles. Requires at least 4 profiles. A subdivide value of 4, means, create 4 face loops between two profiles, without interpolation two profiles create one face loop.

Parameters
• profiles – list of profiles
• subdivide – count of face loops
• close – close profile polygon if True
• caps – close hull with bottom cap and top cap
• ngons – use ngons for caps if True else subdivide caps into triangles

Returns: MeshTransformer
ezdxf.render.forms.rotation_form(count: int, profile: Iterable[Vertex], angle: float = 2 * pi, axis: Vertex = (1, 0, 0)) → MeshTransformer
Create MESH entity by rotating a profile around an axis.

Parameters
• count – count of rotated profiles
• profile – profile to rotate as list of vertices
• angle – rotation angle in radians
• axis – rotation axis

Returns: MeshTransformer

6.9.7 MeshBuilder
The MeshBuilder is a helper class to create Mesh entities. Stores a list of vertices, a list of edges where an edge is a list of indices into the vertices list, and a faces list where each face is a list of indices into the vertices list.
The MeshBuilder.render() method, renders the mesh into a Mesh entity. The Mesh entity supports ngons in AutoCAD, ngons are polygons with more than 4 vertices.
The basic MeshBuilder class does not support transformations.
class ezdxf.render.MeshBuilder

vertices
List of vertices as Vec3 or (x, y, z) tuple

edges
List of edges as 2-tuple of vertex indices, where a vertex index is the index of the vertex in the vertices list.

faces
List of faces as list of vertex indices, where a vertex index is the index of the vertex in the vertices list. A face requires at least three vertices, Mesh supports ngons, so the count of vertices is not limited.

copy()
Returns a copy of mesh.

faces_as_vertices() → Iterable[List[Vec3]]
Iterate over all mesh faces as list of vertices.

edges_as_vertices() → Iterable[Tuple[Vec3, Vec3]]
Iterate over all mesh edges as tuple of two vertices.

add_vertices(vertices: Iterable[Vertex]) → Sequence[int]
Add new vertices to the mesh, each vertex is a (x, y, z) tuple or a Vec3 object, returns the indices of the vertices added to the vertices list.

e.g. adding 4 vertices to an empty mesh, returns the indices (0, 1, 2, 3), adding additional 4 vertices returns the indices (4, 5, 6, 7).

Parameters
vertices – list of vertices, vertex as (x, y, z) tuple or Vec3 objects

Returns
indices of the vertices added to the vertices list

Return type
tuple

add_edge(vertices: Iterable[Vertex]) → None
An edge consist of two vertices [v1, v2], each vertex is a (x, y, z) tuple or a Vec3 object. The new vertex indices are stored as edge in the edges list.

Parameters
vertices – list of 2 vertices : [(x1, y1, z1), (x2, y2, z2)]

add_face(vertices: Iterable[Vertex]) → None
Add a face as vertices list to the mesh. A face requires at least 3 vertices, each vertex is a (x, y, z) tuple or Vec3 object. The new vertex indices are stored as face in the faces list.

Parameters
vertices – list of at least 3 vertices [(x1, y1, z1), (x2, y2, z2), (x3, y3, z3), ...]

add_mesh(vertices=None, faces=None, edges=None, mesh=None) → None
Add another mesh to this mesh.

A mesh can be a MeshBuilder, MeshVertexMerger or Mesh object or requires the attributes vertices, edges and faces.

Parameters

• vertices – list of vertices, a vertex is a (x, y, z) tuple or Vec3 object

• faces – list of faces, a face is a list of vertex indices

• edges – list of edges, an edge is a list of vertex indices

• mesh – another mesh entity
has_none_planar_faces() → bool
Returns True if any face is none planar.

render_mesh(layout: BaseLayout, dxfattribs: dict = None, matrix: Matrix44 = None, ucs: UCS = None)
Render mesh as Mesh entity into layout.

Parameters
• layout – BaseLayout object
• dxfattribs – dict of DXF attributes e.g. {'layer': 'mesh', 'color': 7}
• matrix – transformation matrix of type Matrix44
• ucs – transform vertices by UCS to WCS

render_polyface(layout: BaseLayout, dxfattribs: dict = None, matrix: Matrix44 = None, ucs: UCS = None)
Render mesh as PolyFace entity into layout.

Parameters
• layout – BaseLayout object
• dxfattribs – dict of DXF attributes e.g. {'layer': 'mesh', 'color': 7}
• matrix – transformation matrix of type Matrix44
• ucs – transform vertices by UCS to WCS

render_3dfaces(layout: BaseLayout, dxfattribs: dict = None, matrix: Matrix44 = None, ucs: UCS = None)
Render mesh as Face3d entities into layout.

Parameters
• layout – BaseLayout object
• dxfattribs – dict of DXF attributes e.g. {'layer': 'mesh', 'color': 7}
• matrix – transformation matrix of type Matrix44
• ucs – transform vertices by UCS to WCS

render_normals(layout: BaseLayout, length: float = 1, relative=True, dxfattribs: dict = None)
Render face normals as Line entities into layout, useful to check orientation of mesh faces.

Parameters
• layout – BaseLayout object
• length – visual length of normal, use length < 0 to point normals in opposite direction
• relative – scale length relative to face size if True
• dxfattribs – dict of DXF attributes e.g. {'layer': 'normals', 'color': 6}

classmethod from_mesh(other) → ezdxf.render.mesh.MeshBuilder
Create new mesh from other mesh as class method.

Parameters other – mesh of type MeshBuilder and inherited or DXF Mesh entity or any object providing attributes vertices, edges and faces.
### 6.9.8 MeshTransformer

Same functionality as `MeshBuilder` but supports inplace transformation.

#### ezdxf.render.MeshTransformer

Subclass of `MeshBuilder`

- **subdivide** *(level: int = 1, quads=True, edges=False) → MeshTransformer*
  
  Returns a new `MeshTransformer` object with subdivided faces and edges.
  
  **Parameters**
  
  - **level** – subdivide levels from 1 to max of 5
  - **quads** – create quad faces if `True` else create triangles
  - **edges** – also subdivide edges if `True`

- **transform** *(matrix: Matrix44)*
  
  Transform mesh inplace by applying the transformation `matrix`.
  
  **Parameters**
  
  - **matrix** – 4x4 transformation matrix as `Matrix44` object

- **translate** *(dx: float = 0, dy: float = 0, dz: float = 0)*
  
  Translate mesh inplace.
  
  **Parameters**
  
  - **dx** – translation in x-axis
  - **dy** – translation in y-axis
  - **dz** – translation in z-axis

- **scale** *(sx: float = 1, sy: float = 1, sz: float = 1)*
  
  Scale mesh inplace.
  
  **Parameters**
  
  - **sx** – scale factor for x-axis
  - **sy** – scale factor for y-axis
  - **sz** – scale factor for z-axis

- **scale_uniform** *(s: float)*
  
  Scale mesh uniform inplace.
  
  **Parameters**
  
  - **s** – scale factor for x-, y- and z-axis

- **rotate_x** *(angle: float)*
  
  Rotate mesh around x-axis about `angle` inplace.
  
  **Parameters**
  
  - **angle** – rotation angle in radians

- **rotate_y** *(angle: float)*
  
  Rotate mesh around y-axis about `angle` inplace.
  
  **Parameters**
  
  - **angle** – rotation angle in radians
rotate_z (angle: float)
    Rotate mesh around z-axis about angle inplace.

Parameters
    angle – rotation angle in radians

rotate_axis (axis: Vertex, angle: float)
    Rotate mesh around an arbitrary axis located in the origin (0, 0, 0) about angle.

Parameters
    • axis – rotation axis as Vec3
    • angle – rotation angle in radians

6.9.9 MeshVertexMerger

Same functionality as MeshBuilder, but created meshes with unique vertices and no doublets, but MeshVertexMerger needs extra memory for bookkeeping and also does not support transformations. Location of merged vertices is the location of the first vertex with the same key.

This class is intended as intermediate object to create a compact meshes and convert them to MeshTransformer objects to apply transformations to the mesh:

```python
mesh = MeshVertexMerger()

# create your mesh
mesh.add_face(...)

# convert mesh to MeshTransformer object
return MeshTransformer.from_builder(mesh)
```

class ezdxf.render.MeshVertexMerger (precision: int = 6)
 Mesh with unique vertices and no doublets, but needs extra memory for bookkeeping.

MeshVertexMerger creates a key for every vertex by rounding its components by the Python round() function and a given precision value. Each vertex with the same key gets the same vertex index, which is the index of first vertex with this key, so all vertices with the same key will be located at the location of this first vertex. If you want an average location of and for all vertices with the same key look at the MeshAverageVertexMerger class.

Parameters precision – floating point precision for vertex rounding

6.9.10 MeshAverageVertexMerger

This is an extended version of MeshVertexMerger. Location of merged vertices is the average location of all vertices with the same key, this needs extra memory and runtime in comparision to MeshVertexMerger and this class also does not support transformations.

class ezdxf.render.MeshAverageVertexMerger (precision: int = 6)
 Mesh with unique vertices and no doublets, but needs extra memory for bookkeeping and runtime for calculation of average vertex location.

MeshAverageVertexMerger creates a key for every vertex by rounding its components by the Python round() function and a given precision value. Each vertex with the same key gets the same vertex index, which is the index of first vertex with this key, the difference to the MeshVertexMerger class is the calculation of average vertex location.
the average location for all vertices with the same key, this needs extra memory to keep track of the count of vertices for each key and extra runtime for updating the vertex location each time a vertex with an existing key is added.

**Parameters**
- **precision** – floating point precision for vertex rounding

### 6.9.11 Trace

This module provides tools to create banded lines like LWPOLYLINE with width information. Path rendering as quadrilaterals: **Trace, Solid** or **Face3d**.

**class** *ezdxf.render.trace.TraceBuilder*

Sequence of 2D banded lines like polylines with start- and end width or curves with start- and end width.

Accepts 3D input, but z-axis is ignored.

- **abs_tol**
  Absolute tolerance for floating point comparisons

- **append**(trace: *ezdxf.render.trace.AbstractTrace*) → None
  Append a new trace.

- **close**()
  Close multi traces by merging first and last trace, if linear traces.

- **faces**() → Iterable[Tuple[Vec2, Vec2, Vec2, Vec2]]
  Yields all faces as 4-tuples of Vec2 objects.

- **virtual_entities**(dxftype='TRACE', dxfattribs: Dict[KT, VT] = None, doc: Drawing = None) → Union[Solid, Trace, Face3d]
  Yields faces as SOLID, TRACE or 3DFACE entities with DXF attributes given in dxfattribs.

  If a document is given, the doc attribute of the new entities will be set and the new entities will be automatically added to the entity database of that document.

**Parameters**
- **dxftype** – DXF type as string, “SOLID”, “TRACE” or “3DFACE”
- **dxfattribs** – DXF attributes for SOLID, TRACE or 3DFACE entities
- **doc** – associated document

**classmethod from_polyline**(polyline: DXFGraphic, segments: int = 64) → TraceBuilder

Create a complete trace from a LWPOLYLINE or a 2D POLYLINE entity, the trace consist of multiple sub-traces if bulge values are present.

**Parameters**
- **polyline** – LWPolyline or 2D Polyline
- **segments** – count of segments for bulge approximation, given count is for a full circle, partial arcs have proportional less segments, but at least 3

**class** *ezdxf.render.trace.LinearTrace*

Linear 2D banded lines like polylines with start- and end width.

Accepts 3D input, but z-axis is ignored.
abs_tol
Absolute tolerance for floating point comparisons

is_started
True if at least one station exist.

add_station (point: Vertex, start_width: float, end_width: float = None) → None
Add a trace station (like a vertex) at location point, start_width is the width of the next segment starting at
this station, end_width is the end width of the next segment.

Adding the last location again, replaces the actual last location e.g. adding lines (a, b), (b, c), creates only
3 stations (a, b, c), this is very important to connect to/from splines.

Parameters

• point – 2D location (vertex), z-axis of 3D vertices is ignored.
• start_width – start width of next segment
• end_width – end width of next segment

faces () → Iterable[Tuple[Vec2, Vec2, Vec2, Vec2]]
Yields all faces as 4-tuples of Vec2 objects.

First and last miter is 90 degrees if the path is not closed, otherwise the intersection of first and last segment
is taken into account, a closed path has to have explicit the same last and first vertex.

virtual_entities (dxftype='TRACE', dxfattribs: Dict[KT, VT] = None, doc: Drawing = None) →
Union[Solid, Trace, Face3d]
Yields faces as SOLID, TRACE or 3DFACE entities with DXF attributes given in dxfattribs.

If a document is given, the doc attribute of the new entities will be set and the new entities will be automatically added to the entity database of that document.

Parameters

• dxftype – DXF type as string, “SOLID”, “TRACE” or “3DFACE”
• dxfattribs – DXF attributes for SOLID, TRACE or 3DFACE entities
• doc – associated document

class ezdxf.render.trace.CurvedTrace
2D banded curves like arcs or splines with start- and end width.

Represents always only one curved entity and all miter of curve segments are perpendicular to curve tangents.

Accepts 3D input, but z-axis is ignored.

faces () → Iterable[Tuple[Vec2, Vec2, Vec2, Vec2]]
Yields all faces as 4-tuples of Vec2 objects.

virtual_entities (dxftype='TRACE', dxfattribs: Dict[KT, VT] = None, doc: Drawing = None) →
Union[Solid, Trace, Face3d]
Yields faces as SOLID, TRACE or 3DFACE entities with DXF attributes given in dxfattribs.

If a document is given, the doc attribute of the new entities will be set and the new entities will be automatically added to the entity database of that document.

Parameters

• dxftype – DXF type as string, “SOLID”, “TRACE” or “3DFACE”
• dxfattribs – DXF attributes for SOLID, TRACE or 3DFACE entities
• doc – associated document
classmethod from_arc(arc: ezdxf.math.arc.ConstructionArc, start_width: float, end_width: float, segments: int = 64) → ezdxf.render.trace.CurvedTrace
Create curved trace from an arc.

Parameters

- **arc** – ConstructionArc object
- **start_width** – start width
- **end_width** – end width
- **segments** – count of segments for full circle (360 degree) approximation, partial arcs have proportional less segments, but at least 3

Raises ValueError – if arc.radius <= 0

classmethod from_spline(spline: ezdxf.math.bspline.BSpline, start_width: float, end_width: float, segments: int) → ezdxf.render.trace.CurvedTrace
Create curved trace from a B-spline.

Parameters

- **spline** – BSpline object
- **start_width** – start width
- **end_width** – end width
- **segments** – count of segments for approximation

6.9.12 Point Rendering

Helper function to render Point entities as DXF primitives.

ezdxf.render.point.virtual_entities(point: Point, pdsize: float = 1, pdmode: int = 0) → List[DXFGraphic]
Yields point graphic as DXF primitives LINE and CIRCLE entities. The dimensionless point is rendered as zero-length line!

Check for this condition:

```python
if e.dxftype() == 'LINE' and e.dxf.start.isclose(e.dxf.end)
```

Parameters

- **point** – DXF POINT entity
- **pdsize** – point size in drawing units
- **pdmode** – point styling mode, see Point class

New in version 0.15.

See also:

Go to ezdxf.entities.Point class documentation for more information about POINT styling modes.
6.10 Add-ons

6.10.1 Drawing / Export Addon

This add-on provides the functionality to render a DXF document to produce a rasterized or vector-graphic image which can be saved to a file or viewed interactively depending on the backend being used.

The module provides two example scripts in the folder examples/addons/drawing which can be run to save rendered images to files or view an interactive visualisation

Example for the usage of the matplotlib backend:

```python
import sys
import matplotlib.pyplot as plt
from ezdxf import recover
from ezdxf.addons.drawing import RenderContext, Frontend
from ezdxf.addons.drawing.matplotlib import MatplotlibBackend

# Safe loading procedure (requires ezdxf v0.14):
try:
    doc, auditor = recover.readfile('your.dxf')
except IOError:
    print(f'Not a DXF file or a generic I/O error.
     sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file.')
    sys.exit(2)

# The auditor.errors attribute stores severe errors,
# which may raise exceptions when rendering.
if not auditor.has_errors:
    fig = plt.figure()
    ax = fig.add_axes([0, 0, 1, 1])
    ctx = RenderContext(doc)
    out = MatplotlibBackend(ax)
    Frontend(ctx, out).draw_layout(doc.modelspace(), finalize=True)
    fig.savefig('your.png', dpi=300)
```

Simplified render workflow but with less control:

```python
from ezdxf import recover
from ezdxf.addons.drawing import matplotlib

# Exception handling left out for compactness:
doc, auditor = recover.readfile('your.dxf')
if not auditor.has_errors:
    matplotlib.qsave(doc.modelspace(), 'your.png')
```

MatplotlibBackend

```python
class ezdxf.addons.drawing.matplotlib.MatplotlibBackend

    __init__(ax: plt.Axes, *, adjust_figure: bool = True, font: FontProperties, use_text_cache: bool = True, params: Dict = None)
```
PyQtBackend

class ezdxf.addons.drawing.pyqt.PyQtBackend

    __init__(scene: qw.QGraphicsScene = None, *, use_text_cache: bool = True, debug_draw_rect: bool = False, params: Dict = None)

Backend Options params

Additional options for a backend can be passed by the `params` argument of the backend constructor `__init__()`. Not every option will be supported by all backends and currently most options are only supported by the Matplotlib backend.

**pdsize**  size for the POINT entity:
  - 0 for 5% of draw area height
  - < 0 specifies a percentage of the viewport size
  - > 0 specifies an absolute size

**pdmode**  see `Point` class documentation

**linetype_renderer**
  - “internal” uses the Matplotlib linetype renderer which is oriented on the output medium and dpi setting. This method is simpler and faster but may not replicate the results of CAD applications.
  - “ezdxf” replicate AutoCAD linetype rendering oriented on drawing units and various ltscale factors. This rendering method break lines into small segments which causes a longer rendering time!

**linetype_scaling**  Overall linetype scaling factor. Set to 0 to disable linetype support at all.

**lineweight_scaling**  Overall lineweight scaling factor. Set to 0 to disable lineweight support at all. The current result is correct, in SVG the line width is 0.7 points for 0.25mm as required, but this often looks too thick.

**min_lineweight**  Minimum lineweight.

**min_dash_length**  Minimum dash length.

**max_flattening_distance**  Maximum flattening distance in drawing units for curve approximations.

**show_defpoints**
  - 0 to disable defpoints (default)
  - 1 to show defpoints

**show_hatch**
  - 0 to disable HATCH entities
  - 1 to show HATCH entities

**hatch_pattern**
  - 0 to disable hatch pattern
  - 1 to use predefined Matplotlib pattern by pattern-name matching, or a simplified pattern in the PyQt backend. The PyQt support for hatch pattern is not good, it is often better to turn hatch pattern support off and disable HATCHES by setting `show_hatch` to 0 or use a solid filling.
  - 2 to draw HATCH pattern as solid fillings.
Default Values

<table>
<thead>
<tr>
<th>Backend Option</th>
<th>MatplotlibBackend</th>
<th>PyQtBackend</th>
</tr>
</thead>
<tbody>
<tr>
<td>point_size</td>
<td>2.0</td>
<td>1.0</td>
</tr>
<tr>
<td>point_size_relative</td>
<td>True</td>
<td>not supported</td>
</tr>
<tr>
<td>linetype_renderer</td>
<td>“internal”</td>
<td>“internal”</td>
</tr>
<tr>
<td>linetype_scaling</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>linewidth_scaling</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>min_linewidth</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td>min_dash_length</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>max_flattening_distance</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>show_hatch</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>hatch_pattern</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Properties

class ezdxf.addons.drawing.properties.Properties

LayerProperties

class ezdxf.addons.drawing.properties.LayerProperties

RenderContext

class ezdxf.addons.drawing.properties.RenderContext

Frontend

class ezdxf.addons.drawing.frontend.Frontend

Backend

class ezdxf.addons.drawing.backendBackend

Details

The rendering is performed in two stages. The front-end traverses the DXF document structure, converting each encountered entity into primitive drawing commands. These commands are fed to a back-end which implements the interface: Backend.

Currently a PyQtBackend (QGraphicsScene based) and a MatplotlibBackend are implemented.

Although the resulting images will not be pixel-perfect with AutoCAD (which was taken as the ground truth when developing this add-on) great care has been taken to achieve similar behavior in some areas:

- The algorithm for determining color should match AutoCAD. However, the color palette is not stored in the dxf file, so the chosen colors may be different to what is expected. The RenderContext class supports passing a plot style table (CTB-file) as custom color palette but uses the same palette as AutoCAD by default.
• Text rendering is quite accurate, text positioning, alignment and word wrapping are very faithful. Differences may occur if a different font from what was used by the CAD application but even in that case, for supported backends, measurements are taken of the font being used to match text as closely as possible.

• Visibility determination (based on which layers are visible) should match AutoCAD

See examples/addons/drawing/cad_viewer.py for an advanced use of the module.

See examples/addons/drawing/draw_cad.py for a simple use of the module.

See drawing.md in the ezdxf repository for additional behaviours documented during the development of this add-on.

Limitations

• Line types and hatch patterns/gradients are ignored by the PyQtBackend

• Rich text formatting is ignored (drawn as plain text)

• If the backend does not match the font then the exact text placement and wrapping may appear slightly different

• No support for MULTILEADER

• The style which POINT entities are drawn in are not stored in the dxf file and so cannot be replicated exactly

• only basic support for:
  – infinite lines (rendered as lines with a finite length)
  – viewports (rendered as rectangles)
  – 3D (some entities may not display correctly in 3D (see possible improvements below)) however many things should already work in 3D.
  – vertical text (will render as horizontal text)
  – multiple columns of text (placement of additional columns may be incorrect)

Future Possible Improvements

• pass the font to backend if available

• text formatting commands could be interpreted and broken into text chunks which can be drawn with a single font weight or modification such as italics

6.10.2 Geo Interface

Intended Usage

The intended usage of the ezdxf.addons.geo module is as tool to work with geospatial data in conjunction with dedicated geospatial applications and libraries and the module can not and should not replicate their functionality.

The only reimplemented feature is the most common WSG84 EPSG:3395 World Mercator projection, for everything else use the dedicated packages like:

• pyproj - Cartographic projections and coordinate transformations library.

• Shapely - Manipulation and analysis of geometric objects in the Cartesian plane.

• PyShp - The Python Shapefile Library (PyShp) reads and writes ESRI Shapefiles in pure Python.
• GeoJSON - GeoJSON interface for Python.
• GDAL - Tools for programming and manipulating the GDAL Geospatial Data Abstraction Library.
• Fiona - Fiona is GDAL’s neat and nimble vector API for Python programmers.
• QGIS - A free and open source geographic information system.
• and many more …

This module provides support for the __geo_interface__: https://gist.github.com/sgillies/2217756
Which is also supported by Shapely, for supported types see the GeoJSON Standard and examples in Appendix-A.

See also:
Tutorial for the Geo Add-on for loading GPX data into DXF files with an existing geo location reference and exporting DXF entities as GeoJSON data.

Proxy From Mapping

The GeoProxy represents a __geo_interface__ mapping, create a new proxy by GeoProxy.parse() from an external __geo_interface__ mapping. GeoProxy.to_dxf_entities() returns new DXF entities from this mapping. Returns “Point” as Point entity, “LineString” as LWPolyline entity and “Polygon” as Hatch entity or as separated LWPolyline entities (or both). Supports “MultiPoint”, “MultiLineString”, “MultiPolygon”, “GeometryCollection”, “Feature” and “FeatureCollection”. Add new DXF entities to a layout by the Layout.add_entity() method.

Proxy From DXF Entity

The proxy() function or the constructor GeoProxy.from_dxf_entities() creates a new GeoProxy object from a single DXF entity or from an iterable of DXF entities, entities without a corresponding representation will be approximated.

Supported DXF entities are:

• POINT as “Point”
• LINE as “LineString”
• LWPOLYLINE as “LineString” if open and “Polygon” if closed
• POLYLINE as “LineString” if open and “Polygon” if closed, supports only 2D and 3D polylines, POLYMESH and POLYFACE are not supported
• SOLID, TRACE, 3DFACE as “Polygon”
• CIRCLE, ARC, ELLIPSE and SPLINE by approximation as “LineString” if open and “Polygon” if closed
• HATCH as “Polygon”, holes are supported

Warning: This module does no extensive validity checks for “Polygon” objects and because DXF has different requirements for HATCH boundary paths than the GeoJSON Standard, it is possible to create invalid “Polygon” objects. It is recommended to check critical objects by a sophisticated geometry library like Shapely.
Module Functions

ezdxf.addons.geo.proxy(entity: Union[DXFGraphic, Iterable[DXFGraphic]], distance=0.1, force_line_string=False) → GeoProxy

Returns a GeoProxy object.

Parameters

- entity – a single DXF entity or iterable of DXF entities
- distance – maximum flattening distance for curve approximations
- force_line_string – by default this function returns Polygon objects for closed geometries like CIRCLE, SOLID, closed POLYLINE and so on, by setting argument force_line_string to True, this entities will be returned as LineString objects.

ezdxf.addons.geo.dxf_entities(geo_mapping, polygon=1, dxfattribs: Dict = None) → Iterable[DXFGraphic]

Returns __geo_interface__ mappings as DXF entities.

The polygon argument determines the method to convert polygons, use 1 for Hatch entity, 2 for LWPolyline or 3 for both. Option 2 returns for the exterior path and each hole a separated LWPolyline entity. The Hatch entity supports holes, but has no explicit border line.

Yields Hatch always before LWPolyline entities.

The returned DXF entities can be added to a layout by the Layout.add_entity() method.

Parameters

- geo_mapping – __geo_interface__ mapping as dict or a Python object with a __geo_interface__ property
- polygon – method to convert polygons (1-2-3)
- dxfattribs – dict with additional DXF attributes

ezdxf.addons.geo.gfilter(entities: Iterable[DXFGraphic]) → Iterable[DXFGraphic]

Filter DXF entities from iterable entities, which are incompatible to the __geo_reference__ interface.

GeoProxy Class

class ezdxf.addons.geo.GeoProxy(geo_mapping: Dict[KT, VT], places: int = 6)

Stores the __geo_interface__ mapping in a parsed and compiled form.

Stores coordinates as Vec3 objects and represents “Polygon” always as tuple (exterior, holes) even without holes.

The GeoJSON specification recommends 6 decimal places for latitude and longitude which equates to roughly 10cm of precision. You may need slightly more for certain applications, 9 decimal places would be sufficient for professional survey-grade GPS coordinates.

Parameters

- geo_mapping – parsed and compiled __geo_interface__ mapping
- places – decimal places to round for __geo_interface__ export

__geo_interface__

Returns the __geo_interface__ compatible mapping as dict.

geotype

Property returns the top level entity type or None.
**classmethod parse** *(geo_mapping: Dict) → GeoProxy*

Parse and compile a __geo_interface__ mapping as dict or a Python object with a __geo_interface__ property, does some basic syntax checks, converts all coordinates into Vec3 objects, represents “Polygon” always as tuple (exterior, holes) even without holes.

**classmethod from_dxf_entities** *(entity: Union[DXFGraphic, Iterable[DXFGraphic]], distance=0.1, force_line_string=False) → GeoProxy*

Constructor from a single DXF entity or an iterable of DXF entities.

**Parameters**

- **entity** – DXF entity or entities
- **distance** – maximum flattening distance for curve approximations
- **force_line_string** – by default this function returns Polygon objects for closed geometries like CIRCLE, SOLID, closed POLYLINE and so on, by setting argument force_line_string to True, this entities will be returned as LineString objects.

**to_dxf_entities** *(polygon=1, dxfattribs: Dict = None) → Iterable[DXFGraphic]*

Returns stored __geo_interface__ mappings as DXF entities.

The polygon argument determines the method to convert polygons, use 1 for Hatch entity, 2 for LWPolyline or 3 for both. Option 2 returns for the exterior path and each hole a separated LWPolyline entity. The Hatch entity supports holes, but has no explicit border line.

Yields Hatch always before LWPolyline entities.

The returned DXF entities can be added to a layout by the Layout.add_entity() method.

**Parameters**

- **polygon** – method to convert polygons (1-2-3)
- **dxfattribs** – dict with additional DXF attributes

**copy** () → GeoProxy

Returns a deep copy.

**__iter__** () → Iterable[Dict[KT, VT]]

Iterate over all geo content objects.

Yields only “Point”, “LineString”, “Polygon”, “MultiPoint”, “MultiLineString” and “MultiPolygon” objects, returns the content of “GeometryCollection”, “FeatureCollection” and “Feature” as geometry objects (“Point”, . . . ).

**wcs_to_crs** *(crs: Matrix44) → None*

Transform all coordinates recursive from WCS coordinates into Coordinate Reference System (CRS) by transformation matrix crs inplace.

The CRS is defined by the GeoData entity, get the GeoData entity from the modelspace by method get_geodata(). The CRS transformation matrix can be acquired form the GeoData object by get_crs_transformation() method:

```python
doc = ezdxf.readfile('file.dxf')
msp = doc.modelspace()
geodata = msp.get_geodata()
if geodata:
    matrix, axis_ordering = geodata.get_crs_transformation()
```

If axis_ordering is False the CRS is not compatible with the __geo_interface__ or GeoJSON (see chapter 3.1.1).

**Parameters**
crs – transformation matrix of type Matrix44
crs_to_wcs (crs: Matrix44) → None
Transform all coordinates recursive from CRS into WCS coordinates by transformation matrix crs inplace, see also GeoProxy.wcs_to_crs()

Parameters crs -- transformation matrix of type Matrix44

globe_to_map (func: Callable[[Vec3, Vec3]] = None) → None
Transform all coordinates recursive from globe representation in longitude and latitude in decimal degrees into 2D map representation in meters.

Default is WGS84 EPSG:4326 (GPS) to WGS84 EPSG:3395 World Mercator function wgs84_4326_to_3395().

Use the pyproj package to write a custom projection function as needed.

Parameters func -- custom transformation function, which takes one Vec3 object as argument and returns the result as a Vec3 object.

map_to_globe (func: Callable[[Vec3, Vec3]] = None) → None
Transform all coordinates recursive from 2D map representation in meters into globe representation as longitude and latitude in decimal degrees.

Default is WGS84 EPSG:3395 World Mercator to WGS84 EPSG:4326 GPS function wgs84_3395_to_4326().

Use the pyproj package to write a custom projection function as needed.

Parameters func -- custom transformation function, which takes one Vec3 object as argument and returns the result as a Vec3 object.

apply (func: Callable[[Vec3], Vec3]) → None
Apply the transformation function func recursive to all coordinates.

Parameters func -- transformation function as Callable[[Vec3], Vec3]

filter (func: Callable[[ GeoProxy], bool]) → None
Removes all mappings for which func() returns False. The function only has to handle Point, LineString and Polygon entities, other entities like MultiPolygon are divided into separate entities also any collection.

Helper Functions

ezdxf.addons.geo.wgs84_4326_to_3395 (location: Vec3) → Vec3
Transform WGS84 EPSG:4326 location given as latitude and longitude in decimal degrees as used by GPS into World Mercator cartesian 2D coordinates in meters EPSG:3395.

Parameters location -- Vec3 object, x-attribute represents the longitude value (East-West) in decimal degrees and the y-attribute represents the latitude value (North-South) in decimal degrees.

ezdxf.addons.geo.wgs84_3395_to_4326 (location: Vec3, tol=1e-6) → Vec3
Transform WGS84 World Mercator EPSG:3395 location given as cartesian 2D coordinates x, y in meters into WGS84 decimal degrees as longitude and latitude EPSG:4326 as used by GPS.

Parameters

• location -- Vec3 object, z-axis is ignored
• tol -- accuracy for latitude calculation

ezdxf.addons.geo.dms2dd (d: float, m: float = 0, s: float = 0) → float
Convert degree, minutes, seconds into decimal degrees.
ezdxf.addons.geo.dd2dms \(dd: \text{float}\) \(\rightarrow\) Tuple[float, float, float]

Convert decimal degrees into degree, minutes, seconds.

### 6.10.3 Importer

This add-on is meant to import graphical entities from another DXF drawing and their required table entries like LAYER, LTYPE or STYLE.

Because of complex extensibility of the DXF format and the lack of sufficient documentation, I decided to remove most of the possible source drawing dependencies from imported entities, therefore imported entities may not look the same as the original entities in the source drawing, but at least the geometry should be the same and the DXF file does not break.

Removed data which could contain source drawing dependencies: Extension Dictionaries, AppData and XDATA.

**Warning:** DON’T EXPECT PERFECT RESULTS!

The `Importer` supports following data import:

- entities which are really safe to import: LINE, POINT, CIRCLE, ARC, TEXT, SOLID, TRACE, 3DFACE, SHAPE, POLYLINE, ATTRIB, ATTDEF, INSERT, ELLIPSE, MTEXT, LWPOLYLINE, SPLINE, HATCH, MESH, XLINE, RAY, DIMENSION, LEADER, VIEWPORT
- table and table entry import is restricted to LAYER, LTYPE, STYLE, DIMSTYLE
- import of BLOCK definitions is supported
- import of paper space layouts is supported

Import of DXF objects from the OBJECTS section is not supported.

DIMSTYLE override for entities DIMENSION and LEADER is not supported.

Example:

```python
import ezdxf
from ezdxf.addons import Importer

sdoc = ezdxf.readfile('original.dxf')
tdoc = ezdxf.new()

importer = Importer(sdoc, tdoc)

# import all entities from source modelspace into modelspace of the target drawing
importer.import_modelspace()

# import all paperspace layouts from source drawing
importer.import_paperspace_layouts()

# import all CIRCLE and LINE entities from source modelspace into an arbitrary target layout.
# create target layout
tblock = tdoc.blocks.new('SOURCE_ENTS')
# query source entities
ents = sdoc.modelspace().query('CIRCLE LINE')
# import source entities into target block
importer.import_entities(ents, tblock)
```

(continues on next page)
# This is ALWAYS the last & required step, without finalizing the target drawing is
→maybe invalid!
# This step imports all additional required table entries and block definitions.
importer.finalize()

tdoc.saveas('imported.dxf')

class ezdxf.addons.importer.Importer (source: Drawing, target: Drawing)
The Importer class is central element for importing data from other DXF drawings.

Parameters
• source – source Drawing
• target – target Drawing

Variables
• source – source drawing
• target – target drawing
• used_layer – Set of used layer names as string, AutoCAD accepts layer names without
  a LAYER table entry.
• used_linetypes – Set of used linetype names as string, these linetypes require a TECHNICAL TABLE entry or AutoCAD will crash.
• used_styles – Set of used text style names, these text styles require a TABLE entry or
  AutoCAD will crash.
• used_dimstyles – Set of used dimension style names, these dimension styles require a
  TABLE entry or AutoCAD will crash.

finalize () → None
Finalize import by importing required table entries and block definition, without finalization the target
drawing is maybe invalid for AutoCAD. Call finalize() as last step of the import process.

import_block (block_name: str, rename=True) → str
Import one block definition. If block already exist the block will be renamed if argument rename is
True, else the existing target block will be used instead of the source block. Required name resolving for
imported block references (INSERT), will be done in Importer.finalize().

To replace an existing block in the target drawing, just delete it before importing: target.blocks.
delete_block(block_name, safe=False)

Parameters
• block_name – name of block to import
• rename – rename block if exists in target drawing

Returns: block name (renamed)

Raises ValueError – source block not found

import_blocks (block_names: Iterable[str], rename=False) → None
Import all block definitions. If block already exist the block will be renamed if argument rename is
True, else the existing target block will be used instead of the source block. Required name resolving for
imported block references (INSERT), will be done in Importer.finalize().

Parameters
• **block_names** – names of blocks to import

• **rename** – rename block if exists in target drawing

**Raises** ValueError – source block not found

**import_entities** *(entities: Iterable[DXFEntity], target_layout: BaseLayout = None) → None*

Import all entities into target_layout or the modelspace of the target drawing, if target_layout is ‘None’.

**Parameters**

• **entities** – Iterable of DXF entities

• **target_layout** – any layout (modelspace, paperspace or block) from the target drawing

**Raises** DXFStructureError – target_layout is not a layout of target drawing

**import_entity** *(entity: DXFEntity, target_layout: BaseLayout = None) → None*

Imports a single DXF entity into target_layout or the modelspace of the target drawing, if target_layout is None.

**Parameters**

• **entity** – DXF entity to import

• **target_layout** – any layout (modelspace, paperspace or block) from the target drawing

**Raises** DXFStructureError – target_layout is not a layout of target drawing

**import_modelspace** *(target_layout: BaseLayout = None) → None*

Import all entities from source modelspace into target_layout or the modelspace of the target drawing, if target_layout is None.

**Parameters** target_layout – any layout (modelspace, paperspace or block) from the target drawing

**Raises** DXFStructureError – target_layout is not a layout of target drawing

**import_paperspace_layout** *(name: str) → Layout*

Import paperspace layout name into target drawing. Recreates the source paperspace layout in the target drawing, renames the target paperspace if already a paperspace with same name exist and imports all entities from source paperspace into target paperspace.

**Parameters** name – source paper space name as string

**Returns**: new created target paperspace Layout

**Raises**

• KeyError – source paperspace does not exist

• DXFTypeError – invalid modelspace import

**import_paperspace_layouts** *(name: str) → None*

Import all paperspace layouts and their content into target drawing. Target layouts will be renamed if already a layout with same name exist. Layouts will be imported in original tab order.

**import_shape_files** *(fonts: Set[str]) → None*

Import shape file table entries from source drawing into target drawing. Shape file entries are stored in the styles table but without a name.

**import_table** *(name: str, entries: Union[str, Iterable[str]] = '*', replace=False) → None*

Import specific table entries from source drawing into target drawing.
ezdxf Documentation, Release 0.16.2

Parameters

- **name** – valid table names are `layers`, `linetypes` and `styles`  
- **entries** – Iterable of table names as strings, or a single table name or `*` for all table entries  
- **replace** – True to replace already existing table entry else ignore existing entry

Raises **TypeError** – unsupported table type

```python
import_tables (table_names: Union[str, Iterable[str]] = '*', replace=False) → None
```

Import DXF tables from source drawing into target drawing.

Parameters

- **table_names** – iterable of tables names as strings, or a single table name as string or `*` for all supported tables  
- **replace** – True to replace already existing table entries else ignore existing entries

Raises **TypeError** – unsupported table type

```python
recreate_source_layout (name: str) → Layout
```

Recreate source paperspace layout `name` in the target drawing. The layout will be renamed if `name` already exist in the target drawing. Returns target modelspace for layout name “Model”.

Parameters **name** – layout name as string

Raises **KeyError** – if source layout `name` not exist

6.10.4 dxf2code

Translate DXF entities and structures into Python source code.

Short example:

```python
import ezdxf
from ezdxf.addons.dxf2code import entities_to_code, block_to_code

doc = ezdxf.readfile('original.dxf')
msp = doc.modelspace()
source = entities_to_code(msp)

# create source code for a block definition
block_source = block_to_code(doc.blocks['MyBlock'])

# merge source code objects
source.merge(block_source)

with open('source.py', mode='wt') as f:
    f.write(source.import_str())
    f.write('

')
    f.write(source.code_str())
    f.write('

')
```

```python
ezdxf.addons.dxf2code.entities_to_code (entities: Iterable[DXFEntity], layout: str = 'layout', ignore: Iterable[None] = None) → Code
```

Translates DXF entities into Python source code to recreate this entities by ezdxf.

Parameters
• **entities** – iterable of DXFEntity
• **layout** – variable name of the layout (model space or block) as string
• **ignore** – iterable of entities types to ignore as strings like ['IMAGE', 'DIMENSION']

Returns **Code**


Translates a BLOCK into Python source code to recreate the BLOCK by ezdxf.

Parameters

• **block** – block definition layout
• **drawing** – variable name of the drawing as string
• **ignore** – iterable of entities types to ignore as strings like ['IMAGE', 'DIMENSION']

Returns **Code**

ezdxf.addons.dxf2code.table_entries_to_code(entities: Iterable[DXFEntity], drawing='doc') → Code

**class** ezdxf.addons.dxf2code.Code

Source code container.

**code**

Source code line storage, store lines without line ending \n
**imports**

source code line storage for global imports, store lines without line ending \n
**layers**

Layers used by the generated source code, AutoCAD accepts layer names without a LAYER table entry.

**linetypes**

Linetypes used by the generated source code, these linetypes require a TABLE entry or AutoCAD will crash.

**styles**

Text styles used by the generated source code, these text styles require a TABLE entry or AutoCAD will crash.

**dimstyles**

Dimension styles used by the generated source code, these dimension styles require a TABLE entry or AutoCAD will crash.

**blocks**

Blocks used by the generated source code, these blocks require a BLOCK definition in the BLOCKS section or AutoCAD will crash.

**code_str** *(indent: int = 0) → str*

Returns the source code as a single string.

Parameters **indent** – source code indentation count by spaces

**import_str** *(indent: int = 0) → str*

Returns required imports as a single string.

Parameters **indent** – source code indentation count by spaces

**merge** *(code: ezdxf.addons.dxf2code.Code, indent: int = 0) → None*

Add another **Code** object.
**add_import** *(statement: str) → None*
Add import statement, identical import statements are merged together.

**add_line** *(code: str, indent: int = 0) → None*
Add a single source code line without line ending `\n`.

**add_lines** *(code: Iterable[str], indent: int = 0) → None*
Add multiple source code lines without line ending `\n`.

### 6.10.5 iterdxf

This add-on allows iterating over entities of the modelspace of really big (> 5GB) DXF files which do not fit into memory by only loading one entity at the time. Only ASCII DXF files are supported.

The entities are regular `DXFGraphic` objects with access to all supported DXF attributes, this entities can be written to new DXF files created by the `IterDXF.export()` method. The new `add_foreign_entity()` method allows also to add this entities to new regular `ezdxf` drawings (except for the INSERT entity), but resources like linetype and style are removed, only layer will be preserved but only with default attributes like color 7 and linetype CONTINUOUS.

The following example shows how to split a big DXF files into several separated DXF files which contains only LINE, TEXT or POLYLINE entities.

```python
def add_foreign_entity(line):
    newdoc = ezdxf.new()
    msp = newdoc.modelspace()
    # line is an entity from a big source file
    # and so on ...
```

Supported DXF types:

3DFACE, ARC, ATDDEF, ATTRIB, CIRCLE, DIMENSION, ELLIPSE, HATCH, HELIX, IMAGE, INSERT, LEADER, LINE, LWPOLYLINE, MESH, MLEADER, MLINE, MTEXT, POINT, POLYLINE, Ray, SHAPE, SOLID, SPLINE, TEXT, TRACE, VERTEX, WIPEOUT, XLINE

Transfer simple entities to another DXF document, this works for some supported entities, except for entities with strong dependencies to the original document like INSERT look at `add_foreign_entity()` for all supported types:
 MSP.add_foreign_entity(LinePolyline)
 MSP.add_foreign_entity(mesh)
 MSP.add_foreign_entity(polyface)

Transfer MESH and POLYFACE (dxftype for POLYFACE and POLYMESH is POLYLINE!) entities into a new DXF document by the MeshTransformer class:

```python
from ezdxf.render import MeshTransformer

# mesh is MESH from a big source file
mesh = MeshTransformer.from_mesh(mesh)
# create a new MESH entity from MeshTransformer
MeshTransformer.render(mesh)

# polyface is POLYFACE from a big source file
polyface = MeshTransformer.from_polyface(polyface)
# create a new POLYMESH entity from MeshTransformer
MeshTransformer.render_polyface(polyface)
```

Another way to import entities from a big source file into new DXF documents is to split the big file into smaller parts and use the Importer add-on for a more safe entity import.

```python
ezdxf.addons.iterdxf.opendxf(filename: str, errors: str='surrogateescape') → IterDXF
Open DXF file for iterating, be sure to open valid DXF files, no DXF structure checks will be applied.

Use this function to split up big DXF files as shown in the example above.

Parameters

- **filename** – DXF filename of a seekable DXF file.
- **errors** – specify decoding error handler
  - "surrogateescape" to preserve possible binary data (default)
  - "ignore" to use the replacement char U+FFFD "" for invalid data
  - "strict" to raise an UnicodeDecodeError exception for invalid data

 Raises

- DXFStructureError – invalid or incomplete DXF file
- UnicodeDecodeError – if errors is “strict” and a decoding error occurs

```python
ezdxf.addons.iterdxf.modelspace(filename: str, types:Iterable[str]=None, errors: str='surrogateescape') → Iterable[DXFGraphic]
Iterate over all modelspace entities as DXFGraphic objects of a seekable file.

Use this function to iterate “quick” over modelspace entities of a DXF file, filtering DXF types may speed up things if many entity types will be skipped.

Parameters

- **filename** – filename of a seekable DXF file
- **types** – DXF types like ['LINE', '3DFACE'] which should be returned, None returns all supported types.
- **errors** – specify decoding error handler
  - "surrogateescape" to preserve possible binary data (default)
  - "ignore" to use the replacement char U+FFFD "" for invalid data
ezdxf Documentation, Release 0.16.2

– "strict" to raise an UnicodeDecodeError exception for invalid data

Raises

• DXFStructureError – invalid or incomplete DXF file
• UnicodeDecodeError – if errors is “strict” and a decoding error occurs

ezdxf.addons.iterdxf.single_pass_modelspace(stream: BinaryIO, types:Iterable[str]=None, errors: str='surrogatescape') → Iterable[DXFGraphic]

Iterate over all modelspace entities as DXFGraphic objects in one single pass.

Use this function to ‘quick’ iterate over modelspace entities of a not seekable binary DXF stream, filtering DXF types may speed up things if many entity types will be skipped.

Parameters

• stream – (not seekable) binary DXF stream
• types – DXF types like ['LINE', '3DFACE'] which should be returned, None returns all supported types.
• errors – specify decoding error handler
  – “surrogatescape” to preserve possible binary data (default)
  – ”ignore” to use the replacement char U+FFFD “” for invalid data
  – ”strict” to raise an UnicodeDecodeError exception for invalid data

Raises

• DXFStructureError – Invalid or incomplete DXF file
• UnicodeDecodeError – if errors is “strict” and a decoding error occurs

class ezdxf.addons.iterdxf.IterDXFWriter

eexport (name: str) → IterDXFWriter

Returns a companion object to export parts from the source DXF file into another DXF file, the new file will have the same HEADER, CLASSES, TABLES, BLOCKS and OBJECTS sections, which guarantees all necessary dependencies are present in the new file.

Parameters name – filename, no special requirements

modelspace (types: Iterable[str] = None) → Iterable[DXFGraphic]

Returns an iterator for all supported DXF entities in the modelspace. These entities are regular DXFGraphic objects but without a valid document assigned. It is not possible to add these entities to other ezdxf documents.

It is only possible to recreate the objects by factory functions base-on attributes of the source entity. For MESH, POLYMESH and POLYFACE it is possible to use the MeshTransformer class to render (recreate) this objects as new entities in another document.

Parameters types – DXF types like ['LINE', '3DFACE'] which should be returned, None returns all supported types.

close()

Safe closing source DXF file.

class ezdxf.addons.iterdxf.IterDXFWriter
**write** *(entity: DXFGraphic)*

Write a DXF entity from the source DXF file to the export file.

Don’t write entities from different documents than the source DXF file, dependencies and resources will not match, maybe it will work once, but not in a reliable way for different DXF documents.

**close**

Safe closing of exported DXF file. Copying of OBJECTS section happens only at closing the file, without closing the new DXF file is invalid.

### 6.10.6 r12writer

The fast file/stream writer creates simple DXF R12 drawings with just an ENTITIES section. The HEADER, TABLES and BLOCKS sections are not present except FIXED-TABLES are written. Only LINE, CIRCLE, ARC, TEXT, POINT, SOLID, 3DFACE and POLYLINE entities are supported. FIXED-TABLES is a predefined TABLES section, which will be written, if the init argument `fixed_tables` of `R12FastStreamWriter` is True.

The `R12FastStreamWriter` writes the DXF entities as strings direct to the stream without creating an in-memory drawing and therefore the processing is very fast.

Because of the lack of a BLOCKS section, BLOCK/INSERT can not be used. Layers can be used, but this layers have a default setting color = 7 (black/white) and linetype = 'Continuous'. If writing the FIXED-TABLES, some predefined text styles and line types are available, else text style is always 'STANDARD' and line type is always 'ByLayer'.

If using FIXED-TABLES, following predefined line types are available:

- CONTINUOUS
- CENTER ______ __ ___ __ __ __ __ __ __ __
- CENTERX2 __________ __ __ __ __ __ __ __ __
- CENTER2 ______ __ __ __ __ __ __ __ __ __
- DASHED __ __ __ __ __ __ __ __ __ __ __ __ __
- DASHEDX2 ______ __ __ __ __ __ __ __ __ __
- DASHED2 ______ __ __ __ __ __ __ __ __ __ __
- PHANTOM ______ __ __ __ __ __ __ __ __ __
- PHANTOMX2 __________ __ __ __ __ __ __ __ __
- PHANTOM2 ______ __ __ __ __ __ __ __ __ __ __
- DASHDOT __ . __ . __ . __ . __ . __ . __ . __
- DASHDOTX2 ______ . __ __ . __ __ . __ __
- DASHDOT2 ______ . __ __ . __ __ . __ __ . __
- DOT . . . . . . . . . . . . . . . . . . . . . .
- DOTX2 . . . . . . . . . . . . . . . . . . . . . .
- DOT2 . . . . . . . . . . . . . . . . . . . . . .
- DIVIDE __ __ __ __ __ __ __ __ __ __ __ __ __
- DIVIDEX2 ______ __ __ __ __ __ __ __ __ __
- DIVIDE2 ______ __ __ __ __ __ __ __ __ __ __

If using FIXED-TABLES, following predefined text styles are available:
A simple example with different DXF entities:

```python
from random import random
from ezdxf.addons import r12writer

with r12writer("quick_and_dirty_dxf_r12.dxf") as dxf:
    dxf.add_line((0, 0), (17, 23))
    dxf.add_circle((0, 0), radius=2)
    dxf.add_arc((0, 0), radius=3, start=0, end=175)
    dxf.add_solid([[0, 0], [1, 0], [0, 1], [1, 1]])
    dxf.add_point((1.5, 1.5))
    # 2d polyline, new in v0.12
    dxf.add_polyline_2d(((5, 5), (7, 3), (7, 6)))
    # 2d polyline with bulge value, new in v0.12
    dxf.add_polyline_2d(((5, 5), (7, 3, 0.5), (7, 6)), format='xyb')
    # 3d polyline only, changed in v0.12
    dxf.add_polyline(((4, 3, 2), (8, 5, 0), (2, 4, 9)))
    dxf.add_text("test the text entity", align="MIDDLECENTER")
```

A simple example of writing really many entities in a short time:

```python
from random import random
from ezdxf.addons import r12writer

MAX_X_COORD = 1000.0
MAX_Y_COORD = 1000.0
CIRCLE_COUNT = 1000000

with r12writer("many_circles.dxf") as dxf:
    for i in range(CIRCLE_COUNT):
        dxf.add_circle((MAX_X_COORD*random(), MAX_Y_COORD*random()), radius=2)
```

Show all available line types:

```python
import ezdxf

LINETYPES = ['CONTINUOUS', 'CENTER', 'CENTERX2', 'CENTER2',
             'DASHED', 'DASHEDX2', 'DASHED2', 'PHANTOM', 'PHANTOMX2',
             'PHANTOM2', 'DASHDOT', 'DASHDOTX2', 'DASHDOT2', 'DOT',
             'DOTX2', 'DOT2', 'DIVIDE', 'DIVIDEX2', 'DIVIDE2',
            ]

with r12writer('r12_linetypes.dxf', fixed_tables=True) as dxf:
    for n, ltype in enumerate(LINETYPES):
        dxf.add_line((0, n), (10, n), linetype=ltype)
        dxf.add_text(ltype, (0, n+0.1), height=0.25, style='OpenSansCondensed-Light')
```
ezdxf.addons.r12writer.r12writer (stream: Union[TextIO, BinaryIO, str], fixed_tables = False, fmt = 'asc') → R12FastStreamWriter

Context manager for writing DXF entities to a stream/file. stream can be any file like object with a write() method or just a string for writing DXF entities to the file system. If fixed_tables is True, a standard TABLES section is written in front of the ENTITIES section and some predefined text styles and line types can be used.

Set argument fmt to “asc” to write ASCII DXF file (default) or “bin” to write Binary DXF files. ASCII DXF require a TextIO stream and Binary DXF require a BinaryIO stream.

class ezdxf.addons.r12writer.R12FastStreamWriter (stream: [<class 'typing.TextIO'>, <class 'ezdxf.addons.r12writer.BinaryDXFWriter'>], fixed_tables=False)

Fast stream writer to create simple DXF R12 drawings.

Parameters

• stream – a file like object with a write() method.

• fixed_tables – if fixed_tables is True, a standard TABLES section is written in front of the ENTITIES section and some predefined text styles and line types can be used.

close() → None

Writes the DXF tail. Call is not necessary when using the context manager r12writer().

add_line (start: Sequence[float], end: Sequence[float], layer: str = '0', color: int = None, linetype: str = None) → None

Add a LINE entity from start to end.

Parameters

• start – start vertex as (x, y[, z]) tuple

• end – end vertex as (x, y[, z]) tuple

• layer – layer name as string, without a layer definition the assigned color = 7 (black/white) and line type is 'Continuous'.

• color – color as AutoCAD Color Index (ACI) in the range from 0 to 256, 0 is ByBlock and 256 is ByLayer, default is ByLayer which is always color = 7 (black/white) without a layer definition.

• linetype – line type as string, if FIXED-TABLES are written some predefined line types are available, else line type is always ByLayer, which is always 'Continuous' without a LAYERS table.

add_circle (center: Sequence[float], radius: float, layer: str = '0', color: int = None, linetype: str = None) → None

Add a CIRCLE entity.

Parameters

• center – circle center point as (x, y) tuple

• radius – circle radius as float

• layer – layer name as string see add_line()

• color – color as AutoCAD Color Index (ACI) see add_line()

• linetype – line type as string see add_line()
**add_arc** *(center: Sequence[float], radius: float, start: float = 0, end: float = 360, layer: str = '0', color: int = None, linetype: str = None) → None*

Add an ARC entity. The arc goes counter clockwise from *start* angle to *end* angle.

**Parameters**

- **center** – arc center point as *(x, y)* tuple
- **radius** – arc radius as float
- **start** – arc start angle in degrees as float
- **end** – arc end angle in degrees as float
- **layer** – layer name as string see *add_line()*
- **color** – color as AutoCAD Color Index (ACI) see *add_line()*
- **linetype** – line type as string see *add_line()*

**add_point** *(location: Sequence[float], layer: str = '0', color: int = None, linetype: str = None) → None*

Add a POINT entity.

**Parameters**

- **location** – point location as *(x, y [,z])* tuple
- **layer** – layer name as string see *add_line()*
- **color** – color as AutoCAD Color Index (ACI) see *add_line()*
- **linetype** – line type as string see *add_line()*

**add_3dface** *(vertices: Iterable[Sequence[float]], invisible: int = 0, layer: str = '0', color: int = None, linetype: str = None) → None*

Add a 3DFACE entity. 3DFACE is a spatial area with 3 or 4 vertices, all vertices have to be in the same plane.

**Parameters**

- **vertices** – iterable of 3 or 4 *(x, y, z)* vertices.
- **invisible** – bit coded flag to define the invisible edges,
  1. edge = 1
  2. edge = 2
  3. edge = 4
  4. edge = 8
  Add edge values to set multiple edges invisible, 1. edge + 3. edge = 1 + 4 = 5, all edges = 15
- **layer** – layer name as string see *add_line()*
- **color** – color as AutoCAD Color Index (ACI) see *add_line()*
- **linetype** – line type as string see *add_line()*

**add_solid** *(vertices: Iterable[Sequence[float]], layer: str = '0', color: int = None, linetype: str = None) → None*

Add a SOLID entity. SOLID is a solid filled area with 3 or 4 edges and SOLID is a 2D entity.

**Parameters**

- **vertices** – iterable of 3 or 4 *(x, y [,z])* tuples, z-axis will be ignored.
• **layer** – layer name as string see `add_line()`

• **color** – color as *AutoCAD Color Index (ACI)* see `add_line()`

• **linetype** – line type as string see `add_line()`

```python
def add_polyline_2d(points: Iterable[Sequence[T_co]], format: str = 'xy', closed: bool = False, start_width: float = 0, end_width: float = 0, layer: str = '0', color: int = None, linetype: str = None) → None
```

Add a 2D POLYLINE entity with start width, end width and bulge value support.

**Format codes:**

<table>
<thead>
<tr>
<th>x</th>
<th>x-coordinate</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>y-coordinate</td>
</tr>
<tr>
<td>s</td>
<td>start width</td>
</tr>
<tr>
<td>e</td>
<td>end width</td>
</tr>
<tr>
<td>b</td>
<td>bulge value</td>
</tr>
<tr>
<td>v</td>
<td>(x, y) tuple (z-axis is ignored)</td>
</tr>
</tbody>
</table>

**Parameters**

• **points** – iterable of (x, y, [start_width, [end_width, [bulge]]]) tuple, value order according to the **format** string, unset values default to 0

• **format** – format: format string, default is 'xy'

• **closed** – True creates a closed polyline

• **start_width** – default start width, default is 0

• **end_width** – default end width, default is 0

• **layer** – layer name as string see `add_line()`

• **color** – color as *AutoCAD Color Index (ACI)* see `add_line()`

• **linetype** – line type as string see `add_line()`

```python
def add_polyline(vertices: Iterable[Sequence[float]], closed: bool = False, layer: str = '0', color: int = None, linetype: str = None) → None
```

Add a 3D POLYLINE entity.

**Parameters**

• **vertices** – iterable of (x, y [, z]) tuples, z-axis is 0 by default

• **closed** – True creates a closed polyline

• **layer** – layer name as string see `add_line()`

• **color** – color as *AutoCAD Color Index (ACI)* see `add_line()`

• **linetype** – line type as string see `add_line()`

Changed in version 0.12: Write only 3D POLYLINE entity, added **closed** argument.

```python
def add_polyface(vertices: Iterable[Sequence[float]], faces: Iterable[Sequence[int]], layer: str = '0', color: int = None, linetype: str = None) → None
```

Add a POLYFACE entity. The POLYFACE entity supports only faces of maximum 4 vertices, more indices will be ignored. A simple square would be:
v0 = (0, 0, 0)
v1 = (1, 0, 0)
v2 = (1, 1, 0)
v3 = (0, 1, 0)
dxf.add_polyface(vertices=[v0, v1, v2, v3], faces=[(0, 1, 2, 3)])

All 3D form functions of the `ezdxf.render.forms` module return `MeshBuilder` objects, which provide the required vertex and face lists.

See sphere example: https://github.com/mozman/ezdxf/blob/master/examples/r12writer.py

**Parameters**

- **vertices** – iterable of (x, y, z) tuples
- **faces** – iterable of 3 or 4 vertex indices, indices have to be 0-based
- **layer** – layer name as string see `add_line()`
- **color** – color as `AutoCAD Color Index (ACI)` see `add_line()`
- **linetype** – line type as string see `add_line()`

**add_polymesh** (vertices: `Iterable[Sequence[float]]`, size: `Tuple[int, int]`, closed: `Tuple[bool, bool]`, layer: `str = '0'`, color: `int = None`, linetype: `str = None`) → None

Add a POLYMESH entity. A POLYMESH is a mesh of m rows and n columns, each mesh vertex has its own x-, y- and z coordinates. The mesh can be closed in m- and/or n-direction. The vertices have to be in column order: (m0, n0), (m0, n1), (m0, n2), (m1, n0), (m1, n1), (m1, n2), ...

See example: https://github.com/mozman/ezdxf/blob/master/examples/r12writer.py

**Parameters**

- **vertices** – iterable of (x, y, z) tuples, in column order
- **size** – mesh dimension as (m, n)-tuple, requirement: `len(vertices) == m*n`
- **closed** – (m_closed, n_closed) tuple, for closed mesh in m and/or n direction
- **layer** – layer name as string see `add_line()`
- **color** – color as `AutoCAD Color Index (ACI)` see `add_line()`
- **linetype** – line type as string see `add_line()`

**add_text** (text: `str`, insert: `Sequence[float] = (0, 0)`, height: `float = 1.0`, width: `float = 1.0`, align: `str = 'LEFT'`, rotation: `float = 0.0`, oblique: `float = 0.0`, style: `str = 'STANDARD'`, layer: `str = '0'`, color: `int = None`) → None

Add a one line TEXT entity.

**Parameters**

- **text** – the text as string
- **insert** – insert location as (x, y) tuple
- **height** – text height in drawing units
- **width** – text width as factor
- **align** – text alignment, see table below
- **rotation** – text rotation in degrees as float
- **oblique** – oblique in degrees as float, vertical = 0 (default)
• **style** – text style name as string, if FIXED-TABLES are written some predefined text
  styles are available, else text style is always 'STANDARD'.

• **layer** – layer name as string see `add_line()`

• **color** – color as *AutoCAD Color Index (ACI)* see `add_line()`

<table>
<thead>
<tr>
<th>Vert/Horiz</th>
<th>Left</th>
<th>Center</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top</td>
<td>TOP_LEFT</td>
<td>TOP_CENTER</td>
<td>TOP_RIGHT</td>
</tr>
<tr>
<td>Middle</td>
<td>MIDDLE_LEFT</td>
<td>MIDDLE_CENTER</td>
<td>MIDDLE_RIGHT</td>
</tr>
<tr>
<td>Bottom</td>
<td>BOTTOM_LEFT</td>
<td>BOTTOM_CENTER</td>
<td>BOTTOM_RIGHT</td>
</tr>
<tr>
<td>Baseline</td>
<td>LEFT</td>
<td>CENTER</td>
<td>RIGHT</td>
</tr>
</tbody>
</table>

The special alignments ALIGNED and FIT are not available.

### 6.10.7 ODA File Converter Support

Use an installed ODA File Converter for converting between different versions of .dwg, .dxb and .dx.

**Warning:** Execution of an external application is a big security issue! Especially when the path to the executable

To avoid this problem delete the ezdxf.addons.odafc.py module.

The ODA File Converter has to be installed by the user, the application is available for Windows XP, Windows 7 or

At least at Windows the GUI of the ODA File Converter pops up on every call.

ODA File Converter version strings, you can use any of this strings to specify a version, 'R...' and 'AC....'

strings will be automatically mapped to 'ACAD....' strings:

<table>
<thead>
<tr>
<th>ODAFC</th>
<th>ezdxf</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACAD9</td>
<td>not supported</td>
<td>AC1004</td>
</tr>
<tr>
<td>ACAD10</td>
<td>not supported</td>
<td>AC1006</td>
</tr>
<tr>
<td>ACAD12</td>
<td>R12</td>
<td>AC1009</td>
</tr>
<tr>
<td>ACAD13</td>
<td>R13</td>
<td>AC1012</td>
</tr>
<tr>
<td>ACAD14</td>
<td>R14</td>
<td>AC1014</td>
</tr>
<tr>
<td>ACAD2000</td>
<td>R2000</td>
<td>AC1015</td>
</tr>
<tr>
<td>ACAD2004</td>
<td>R2004</td>
<td>AC1018</td>
</tr>
<tr>
<td>ACAD2007</td>
<td>R2007</td>
<td>AC1021</td>
</tr>
<tr>
<td>ACAD2010</td>
<td>R2010</td>
<td>AC1024</td>
</tr>
<tr>
<td>ACAD2013</td>
<td>R2013</td>
<td>AC1027</td>
</tr>
<tr>
<td>ACAD2018</td>
<td>R2018</td>
<td>AC1032</td>
</tr>
</tbody>
</table>

Usage:

```python
from ezdxf.addons import odafc

# Load a DWG file
doc = odafc.readfile('my.dwg')

# Use loaded document like any other ezdxf document
```

(continues on next page)
ezdxf Documentation, Release 0.16.2

(continued from previous page)

```python
print(f'Document loaded as DXF version: {doc.dxfversion}.')
msp = doc.modelspace()
...

# Export document as DWG file for AutoCAD R2018
odafc.export_dwg(doc, 'my_R2018.dwg', version='R2018')
```

ezdxf.addons.odafc.exec_path

Path to installed ODA File Converter executable, default is "C:\Program Files\ODA\ODAFileConverter\ODAFileConverter.exe".

ezdxf.addons.odafc.temp_path

Path to a temporary folder by default the system temp folder defined by environment variable TMP or TEMP.

ezdxf.addons.odafc.readfile(filename: str, version: str = None, audit=False) → Drawing

Use an installed ODA File Converter to convert a DWG/DXB/DXF file into a temporary DXF file and load this file by ezdxf.

Parameters

- **filename** – file to load by ODA File Converter
- **version** – load file as specific DXF version, by default the same version as the source file or if not detectable the latest by ezdxf supported version.
- **audit** – audit source file before loading

ezdxf.addons.odafc.export_dwg(doc: Drawing, filename: str, version: str = None, audit=False, replace=False) → None

Use an installed ODA File Converter to export a DXF document doc as a DWG file.

Saves a temporary DXF file and convert this DXF file into a DWG file by the ODA File Converter. If version is not specified the DXF version of the source document is used.

Parameters

- **doc** – ezdxf DXF document as Drawing object
- **filename** – export filename of DWG file, extension will be changed to “.dwg”
- **version** – export file as specific version, by default the same version as the source document.
- **audit** – audit source file by ODA File Converter at exporting
- **replace** – replace existing DWG file if True

Changed in version 0.15: added replace option

6.10.8 text2path

New in version 0.16.

Tools to convert text strings and text based DXF entities into outer- and inner linear paths as Path objects. These tools depend on the optional Matplotlib package. At the moment only the TEXT and the ATTRIB entity can be converted into paths and hatches.

Don’t expect a 100% match compared to CAD applications.
Text Alignments

The text alignments work the same way as for the `Text` entity:

<table>
<thead>
<tr>
<th>Vertical</th>
<th>Left</th>
<th>Center</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top</td>
<td>TOP_LEFT</td>
<td>TOP_CENTER</td>
<td>TOP_RIGHT</td>
</tr>
<tr>
<td>Middle</td>
<td>MIDDLE_LEFT</td>
<td>MIDDLE_CENTER</td>
<td>MIDDLE_RIGHT</td>
</tr>
<tr>
<td>Bottom</td>
<td>BOTTOM_LEFT</td>
<td>BOTTOM_CENTER</td>
<td>BOTTOM_RIGHT</td>
</tr>
<tr>
<td>Baseline</td>
<td>LEFT</td>
<td>CENTER</td>
<td>RIGHT</td>
</tr>
</tbody>
</table>

The vertical middle alignments (MIDDLE_XXX), center the text vertically in the middle of the uppercase letter “X” (cap height).

Special alignments, where the horizontal alignment is always in the center of the text:

- ALIGNED: text is scaled to match the given `length`, scales x- and y-direction by the same factor.
- FIT: text is scaled to match the given `length`, but scales only in x-direction.
- MIDDLE: insertion point is the center of the total height (cap height + descender height) without scaling, the `length` argument is ignored.

Font Face Definition

A font face is defined by the Matplotlib compatible `FontFace` object by `font-family`, `font-style`, `font-stretch` and `font-weight`.

See also:
- `Font Anatomy`
- `Font Properties`

String Functions

Entity Functions

6.10.9 PyCSG

Constructive Solid Geometry (CSG) is a modeling technique that uses Boolean operations like union and intersection to combine 3D solids. This library implements CSG operations on meshes elegantly and concisely using BSP trees, and is meant to serve as an easily understandable implementation of the algorithm. All edge cases involving overlapping coplanar polygons in both solids are correctly handled.

Example for usage:

```python
import ezdxf
from ezdxf.render.forms import cube, cylinder_2p
from ezdxf.addons.pycsg import CSG

# create new DXF document
doc = ezdxf.new()
msp = doc.modelspace()

# create same geometric primitives as MeshTransformer() objects
cubel = cube()
```
cylinder1 = cylinder_2p(count=32, base_center=(0, -1, 0), top_center=(0, 1, 0),
    radius=.25)

# build solid union
union = CSG(cube1) + CSG(cylinder1)
# convert to mesh and render mesh to modelspace
union.mesh().render(msp, dxfattribs={'color': 1})

# build solid difference
difference = CSG(cube1) - CSG(cylinder1)
# convert to mesh, translate mesh and render mesh to modelspace
difference.mesh().translate(1.5).render(msp, dxfattribs={'color': 3})

# build solid intersection
intersection = CSG(cube1) * CSG(cylinder1)
# convert to mesh, translate mesh and render mesh to modelspace
intersection.mesh().translate(2.75).render(msp, dxfattribs={'color': 5})

doc.saveas('csg.dxf')

This CSG kernel supports only meshes as MeshBuilder objects, which can be created from and converted to DXF Mesh entities.

This CSG kernel is not compatible with ACIS objects like Solid3d, Body, Surface or Region.

Note: This is a pure Python implementation, don’t expect great performance and the implementation is based on an unbalanced BSP tree, so in the case of RecursionError, increase the recursion limit:
import sys

actual_limit = sys.getrecursionlimit()
# default is 1000, increasing too much may cause a seg fault
sys.setrecursionlimit(10000)

... # do the CSG stuff
sys.setrecursionlimit(actual_limit)

CSG works also with spheres, but with really bad runtime behavior and most likely RecursionError exceptions, and use quadrilaterals as body faces to reduce face count by setting argument quads to True.

import ezdxf

from ezdxf.render.forms import sphere, cube
from ezdxf.addons.pycsg import CSG

doc = ezdxf.new()
doc.set_modelspace_vport(6, center=(5, 0))
msp = doc.modelspace()

cube1 = cube().translate(-.5, -.5, -.5)
sphere1 = sphere(count=32, stacks=16, radius=.5, quads=True)

union = (CSG(cube1) + CSG(sphere1)).mesh()
union.render(msp, dxfattribs={'color': 1})

subtract = (CSG(cube1) - CSG(sphere1)).mesh().translate(2.5)
subtract.render(msp, dxfattribs={'color': 3})

intersection = (CSG(cube1) * CSG(sphere1)).mesh().translate(4)
intersection.render(msp, dxfattribs={'color': 5})
Hard Core CSG - Menger Sponge Level 3 vs Sphere

Required runtime on an old Xeon E5-1620 Workstation @ 3.60GHz, with default recursion limit of 1000 on Windows 10:

- CPython 3.8.1 64bit: ~60 seconds,
- pypy3 [PyPy 7.2.0] 32bit: ~6 seconds, and using __slots__ reduced runtime below 5 seconds, yes - pypy is worth a look for long running scripts!

```python
from ezdxf.render.forms import sphere
from ezdxf.addons import MengerSponge
from ezdxf.addons.pycsg import CSG

doc = ezdxf.new()
doc.layers.new('sponge', dxfattribs={'color': 5})
doc.layers.new('sphere', dxfattribs={'color': 6})

doc.set_modelspace_vport(6, center=(5, 0))
msp = doc.modelspace()
```

(continues on next page)
sponge1 = MengerSponge(level=3).mesh()
sphere1 = sphere(count=32, stacks=16, radius=.5, quads=True).translate(.25, .25, 1)
subtract = (CSG(sponge1, meshid=1) - CSG(sphere1, meshid=2))
# get mesh result by id
subtract.mesh(1).render(msp, dxfattribs={'layer': 'sponge'})
subtract.mesh(2).render(msp, dxfattribs={'layer': 'sphere'})
CSG Class

class ezdxf.addons.pycsg.CSG(mesh: MeshBuilder, meshid: int = 0)
Constructive Solid Geometry (CSG) is a modeling technique that uses Boolean operations like union and intersection to combine 3D solids. This class implements CSG operations on meshes.

New 3D solids are created from MeshBuilder objects and results can be exported as MeshTransformer objects to ezdxf by method mesh().

Parameters

- mesh – ezdxf.render.MeshBuilder or inherited object
- meshid – individual mesh ID to separate result meshes, 0 is default

mesh(meshid: int = 0) → MeshTransformer
Returns a ezdxf.render.MeshTransformer object.

Parameters

- meshid – individual mesh ID, 0 is default

union(other: CSG) → CSG
Return a new CSG solid representing space in either this solid or in the solid other. Neither this solid nor the solid other are modified:

\[
\begin{align*}
A \cup B &= A + B \\
A - B &= \text{difference}
\end{align*}
\]

__add__(other: CSG) → CSG

union = A + B

subtract(other: CSG) → CSG

Return a new CSG solid representing space in this solid but not in the solid other. Neither this solid nor the solid other are modified:

\[
\begin{align*}
A \setminus B &= A - B
\end{align*}
\]
**intersect** *(other: CSG) → CSG*

Return a new CSG solid representing space both this solid and in the solid `other`. Neither this solid nor the solid `other` are modified:

```
A.intersect(B)
```

```plaintext
+-------+
|       |
| A     |
| +-----+ = +--+
| +-----+ | +--+
|       | B |
|       |
+-------+
```

**__mul__** *(other: CSG) → CSG*

```
intersection = A * B
```

**inverse() → CSG**

Return a new CSG solid with solid and empty space switched. This solid is not modified.

---

**License**

- Original implementation csg.js, Copyright (c) 2011 Evan Wallace (http://madebyevan.com/), under the MIT license.
- Python port pycsg, Copyright (c) 2012 Tim Knip (http://www.floorplanner.com), under the MIT license.
- Additions by Alex Pletzer (Pennsylvania State University)
- Integration as ezdxf add-on, Copyright (c) 2020, Manfred Moitzi, MIT License.

---

**6.10.10 Plot Style Files (CTB/STB)**

CTB and STB files store plot styles used by AutoCAD and BricsCAD for printing and plotting.

If the plot style table is attached to a Paperspace or the Modelspace, a change of a plot style affects any object that uses that plot style. CTB files contain color dependent plot style tables, STB files contain named plot style tables.

See also:

- Using plot style tables in AutoCAD
- AutoCAD Plot Style Table Editor
- BricsCAD Plot Style Table Editor
- AUTODESK KNOWLEDGE NETWORK: How to install CTB files in AutoCAD

```python
ezdxf.addons.acadctb.load(filename: str) → Union[ColorDependentPlotStyles, NamedPlotStyles]
```

Load the CTB or STB file `filename` from file system.

```python
ezdxf.addons.acadctb.new_ctb() → ColorDependentPlotStyles
```

Create a new CTB file.

Changed in version 0.10: renamed from `new()`
ezdxf.addons.acadctb.new_stb() → NamedPlotStyles
Create a new STB file.

**ColorDependentPlotStyles**

Color dependent plot style table (CTB file), table entries are `PlotStyle` objects.

class ezdxf.addons.acadctb.ColorDependentPlotStyles

- **description**
  Custom description of plot style file.

- **scale_factor**
  Specifies the factor by which to scale non-ISO linetypes and fill patterns.

- **apply_factor**
  Specifies whether or not you want to apply the `scale_factor`.

- **custom_lineweight_display_units**
  Set 1 for showing lineweight in inch in AutoCAD CTB editor window, but lineweights are always defined in millimeters.

- **lineweights**
  Lineweights table as `array.array`

  - `_getitem_(aci: int) → PlotStyle`
    Returns `PlotStyle` for `AutoCAD Color Index (ACI) aci`.

  - `_iter_()` → Iterable[PlotStyle]
    Iterable of all plot styles.

- **new_style**(aci: int, data: dict = None) → PlotStyle
  Set aci to new attributes defined by data dict.

  **Parameters**

  - **aci** – `AutoCAD Color Index (ACI)`
  - **data** – dict of `PlotStyle` attributes: `description`, `color`, `physical_pen_number`, `virtual_pen_number`, `screen`, `linetype`, `lineweight`, `end_style`, `join_style`, `fill_style`

- **get_lineweight**(aci: int)
  Returns the assigned lineweight for `PlotStyle aci` in millimeter.

- **get_lineweight_index**(lineweight: float) → int
  Get index of lineweight in the lineweight table or append lineweight to lineweight table.

- **get_table_lineweight**(index: int) → float
  Returns lineweight in millimeters of lineweight table entry index.

  **Parameters**

  - **index** – lineweight table index = `PlotStyle.lineweight`

  **Returns**

  Lineweight in mm or 0.0 for use entity lineweight

- **set_table_lineweight**(index: int, lineweight: float) → int
  Argument `index` is the lineweight table index, not the `AutoCAD Color Index (ACI)`.

  **Parameters**

  - **index** – lineweight table index = `PlotStyle.lineweight`
  - **lineweight** – in millimeters
save (filename: str) → None
Save CTB file as filename to the file system.

write (stream: BinaryIO) → None
Compress and write CTB file to binary stream.

**NamedPlotStyles**

Named plot style table (STB file), table entries are `PlotStyle` objects.

```python
class ezdxf.addons.acadctb.NamedPlotStyles
```

description
Custom description of plot style file.

scale_factor
Specifies the factor by which to scale non-ISO linetypes and fill patterns.

apply_factor
Specifies whether or not you want to apply the `scale_factor`.

custom_lineweight_display_units
Set 1 for showing lineweight in inch in AutoCAD CTB editor window, but lineweights are always defined in millimeters.

lineweights
Lineweights table as `array.array`

```python
__getitem__ (name: str) → PlotStyle
Returns `PlotStyle` by name.
```

```python
__delitem__ (name: str)
Delete plot style `name`. Plot style 'Normal' is not deletable.
```

```python
__iter__ () → Iterable[str]
Iterable of all plot style names.
```

```python
new_style (name: str, localized_name: str = None, data: dict = None) → PlotStyle
Create new class `PlotStyle name` by attribute dict `data`, replaces existing class `PlotStyle` objects.
```

Parameters

- name – plot style name
- localized_name – name shown in plot style editor, uses `name` if None
- data – dict of `PlotStyle` attributes: description, color, physical_pen_number, virtual_pen_number, screen, linepattern_size, linetype, adaptive_linetype, lineweight, end_style, join_style, fill_style

```python
get_lineweight (name: str)
Returns the assigned lineweight for `PlotStyle name` in millimeter.
```

```python
get_lineweight_index (lineweight: float) → int
Get index of lineweight in the lineweight table or append lineweight to lineweight table.
```

```python
get_table_lineweight (index: int) → float
Returns lineweight in millimeters of lineweight table entry `index`.
```

Parameters

- index – lineweight table index = `PlotStyle.lineweight`

Returns

lineweight in mm or 0.0 for use entity lineweight
**set_table_lineweight** \((index: \text{int}, \text{lineweight}: \text{float}) \rightarrow \text{int}\)

Argument `index` is the lineweight table index, not the *AutoCAD Color Index (ACI)*.

**Parameters**

- `index` – lineweight table index = `PlotStyle.lineweight`
- `lineweight` – in millimeters

**save** \((filename: \text{str}) \rightarrow \text{None}\)

Save STB file as `filename` to the file system.

**write** \((stream: \text{BinaryIO}) \rightarrow \text{None}\)

Compress and write STB file to binary `stream`.

---

**PlotStyle**

**class** `ezdxfl.addons.acadctb.PlotStyle`

**index**
Table index (0-based). (int)

**aci**

*AutoCAD Color Index (ACI)* in range from 1 to 255. Has no meaning for named plot styles. (int)

**description**
Custom description of plot style. (str)

**physical_pen_number**
Specifies physical plotter pen, valid range from 1 to 32 or `AUTOMATIC`. (int)

**virtual_pen_number**
Only used by non-pen plotters and only if they are configured for virtual pens. valid range from 1 to 255 or `AUTOMATIC`. (int)

**screen**
Specifies the color intensity of the plot on the paper, valid range is from 0 to 100. (int)

If you select 100 the drawing will plotted with its full color intensity. In order for screening to work, the *dithering* option must be active.

**linetype**
Overrides the entity linetype, default value is `OBJECT_LINETYPE`. (bool)

**adaptive_linetype**

True if a complete linetype pattern is more important than a correct linetype scaling, default is `True`. (bool)

**linepattern_size**
Line pattern size, default = 0.5. (float)

**lineweight**

Overrides the entity lineWEIGHT, default value is `OBJECT_LINEWEIGHT`. This is an index into the `UserStyles.lineweights` table. (int)

**end_style**
Line end cap style, see table below, default is `END_STYLE_OBJECT` (int)

**join_style**
Line join style, see table below, default is `JOIN_STYLE_OBJECT` (int)
fill_style
Line fill style, see table below, default is \texttt{FILL\_STYLE\_OBJECT (int)}

dithering
Depending on the capabilities of your plotter, dithering approximates the colors with dot patterns. When this option is \texttt{False}, the colors are mapped to the nearest color, resulting in a smaller range of colors when plotting.

Dithering is available only whether you select the object’s color or assign a plot style color.

grayscale
Plot colors in grayscale. (bool)

Default Line Weights

<table>
<thead>
<tr>
<th>#</th>
<th>[mm]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>0.05</td>
</tr>
<tr>
<td>2</td>
<td>0.09</td>
</tr>
<tr>
<td>3</td>
<td>0.10</td>
</tr>
<tr>
<td>4</td>
<td>0.13</td>
</tr>
<tr>
<td>5</td>
<td>0.15</td>
</tr>
<tr>
<td>6</td>
<td>0.18</td>
</tr>
<tr>
<td>7</td>
<td>0.20</td>
</tr>
<tr>
<td>8</td>
<td>0.25</td>
</tr>
<tr>
<td>9</td>
<td>0.30</td>
</tr>
<tr>
<td>10</td>
<td>0.35</td>
</tr>
<tr>
<td>11</td>
<td>0.40</td>
</tr>
<tr>
<td>12</td>
<td>0.45</td>
</tr>
<tr>
<td>13</td>
<td>0.50</td>
</tr>
<tr>
<td>14</td>
<td>0.53</td>
</tr>
<tr>
<td>15</td>
<td>0.60</td>
</tr>
<tr>
<td>16</td>
<td>0.65</td>
</tr>
<tr>
<td>17</td>
<td>0.70</td>
</tr>
<tr>
<td>18</td>
<td>0.80</td>
</tr>
<tr>
<td>19</td>
<td>0.90</td>
</tr>
<tr>
<td>20</td>
<td>1.00</td>
</tr>
<tr>
<td>21</td>
<td>1.06</td>
</tr>
<tr>
<td>22</td>
<td>1.20</td>
</tr>
<tr>
<td>23</td>
<td>1.40</td>
</tr>
<tr>
<td>24</td>
<td>1.58</td>
</tr>
<tr>
<td>25</td>
<td>2.00</td>
</tr>
<tr>
<td>26</td>
<td>2.11</td>
</tr>
</tbody>
</table>

Predefined Values

\texttt{ezdxfs.addons.acadctb.AUTOMATIC}
\texttt{ezdxfs.addons.acadctb.OBJECT\_LINEWEIGHT}
\texttt{ezdxfs.addons.acadctb.OBJECT\_LINETYPE}
\texttt{ezdxfs.addons.acadctb.OBJECT\_COLOR}
ezdxf.addons.acadctb.**OBJECT_COLOR2**

**Line End Style**

- **Line end style:**
  - Use entity end style
  - **END_STYLE_BUTT** 0
  - **END_STYLE_SQUARE** 1
  - **END_STYLE_ROUND** 2
  - **END_STYLE_DIAMOND** 3
  - **END_STYLE_OBJECT** 4

- **Line join style:**
  - Use entity join style
  - **JOIN_STYLE_MITER** 0
  - **JOIN_STYLE_BEVEL** 1
  - **JOIN_STYLE_ROUND** 2
  - **JOIN_STYLE_DIAMOND** 3
  - **JOIN_STYLE_OBJECT** 5
Fill Style

FILL_STYLE_SOLID 64
FILL_STYLE_CHECKERBOARD 65
FILL_STYLE_CROSSHATCH 66
FILL_STYLE_DIAMONDS 67
FILL_STYLE_HORIZONTAL_BARS 68
FILL_STYLE_SLANT_LEFT 69
FILL_STYLE_SLANT_RIGHT 70
FILL_STYLE_SQUARE_DOTS 71
FILL_STYLE_VERICAL_BARS 72
FILL_STYLE_OBJECT 73
Linetypes

Linetype: Use entity linetype

Linetype: Solid

Lineweight: Dashed

Line end style: Dotted

Line join style: Dash Dot

Fill style: Short Dash

Fill style: Medium Dash

Fill style: Long Dash

Fill style: Short Dash x2

Fill style: Medium Dash x2

Fill style: Long Dash x2

Fill style: Medium Long Dash

Fill style: Medium Dash Short Dash Short Dash

Fill style: Long Dash Short Dash

Fill style: Long Dash Dot Dot

Fill style: Long Dash Dot

Fill style: Long Dash Dot

Fill style: Medium Dash Dot Short Dash Dot

Fill style: Sparse Dot

Fill style: ISO Dash
<table>
<thead>
<tr>
<th>Linetype name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solid</td>
<td>0</td>
</tr>
<tr>
<td>Dashed</td>
<td>1</td>
</tr>
<tr>
<td>Dotted</td>
<td>2</td>
</tr>
<tr>
<td>Dash Dot</td>
<td>3</td>
</tr>
<tr>
<td>Short Dash</td>
<td>4</td>
</tr>
<tr>
<td>Medium Dash</td>
<td>5</td>
</tr>
<tr>
<td>Long Dash</td>
<td>6</td>
</tr>
<tr>
<td>Short Dash x2</td>
<td>7</td>
</tr>
<tr>
<td>Medium Dash x2</td>
<td>8</td>
</tr>
<tr>
<td>Long Dash x2</td>
<td>9</td>
</tr>
<tr>
<td>Medium Lang Dash</td>
<td>10</td>
</tr>
<tr>
<td>Medium Dash Short Dash Short Dash</td>
<td>11</td>
</tr>
<tr>
<td>Long Dash Short Dash</td>
<td>12</td>
</tr>
<tr>
<td>Long Dash Dot Dot</td>
<td>13</td>
</tr>
<tr>
<td>Long Dash Dot</td>
<td>14</td>
</tr>
<tr>
<td>Medium Dash Dot Short Dash Dot</td>
<td>15</td>
</tr>
<tr>
<td>Sparse Dot</td>
<td>16</td>
</tr>
<tr>
<td>ISO Dash</td>
<td>17</td>
</tr>
<tr>
<td>ISO Dash Space</td>
<td>18</td>
</tr>
<tr>
<td>ISO Long Dash Dot</td>
<td>19</td>
</tr>
<tr>
<td>ISO Long Dash Double Dot</td>
<td>20</td>
</tr>
<tr>
<td>ISO Long Dash Triple Dot</td>
<td>21</td>
</tr>
</tbody>
</table>

Continued on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Linetype name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO Dot</td>
<td>22</td>
</tr>
<tr>
<td>ISO Long Dash Short Dash</td>
<td>23</td>
</tr>
<tr>
<td>ISO Long Dash Double Short Dash</td>
<td>24</td>
</tr>
<tr>
<td>ISO Dash Dot</td>
<td>25</td>
</tr>
<tr>
<td>ISO Double Dash Dot</td>
<td>26</td>
</tr>
<tr>
<td>ISO Dash Double Dot</td>
<td>27</td>
</tr>
<tr>
<td>ISO Double Dash Double Dot</td>
<td>28</td>
</tr>
<tr>
<td>ISO Dash Triple Dot</td>
<td>29</td>
</tr>
<tr>
<td>ISO Double Dash Triple Dot</td>
<td>30</td>
</tr>
<tr>
<td>Use entity linetype</td>
<td>31</td>
</tr>
</tbody>
</table>

### 6.10.11 Showcase Forms

#### MengerSponge

Build a 3D Menger sponge.

```python
class ezdxm.addons.MengerSponge:
    location: Vertex = (0.0, 0.0, 0.0), length: float = 1.0, level: int = 1, kind: int = 0
```

**Parameters**

- **location** – location of lower left corner as (x, y, z) tuple
- **length** – side length
- **level** – subdivide level
- **kind** – type of menger sponge

<table>
<thead>
<tr>
<th>Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Original Menger Sponge</td>
</tr>
<tr>
<td>1</td>
<td>Variant XOX</td>
</tr>
<tr>
<td>2</td>
<td>Variant QXO</td>
</tr>
<tr>
<td>3</td>
<td>Jerusalem Cube</td>
</tr>
</tbody>
</table>

**render**

```python
render(layout: GenericLayoutType, merge: bool = False, dxfattribs: dict = None, matrix: Matrix44 = None, ucs: UCS = None) → None
```

Renders the menger sponge into layout, set `merge` to `True` for rendering the whole menger sponge into one MESH entity, set `merge` to `False` for rendering the individual cubes of the menger sponge as MESH entities.

**Parameters**

- **layout** – DXF target layout
- **merge** – `True` for one MESH entity, `False` for individual MESH entities per cube
- **dxfattribs** – DXF attributes for the MESH entities
- **matrix** – apply transformation matrix at rendering
- **ucs** – apply UCS transformation at rendering

**cubes**

```python
cubes() → Iterable[ezdxm.render.mesh.MeshTransformer]
```

Yields all cubes of the menger sponge as individual `MeshTransformer` objects.
mesh() \rightarrow \texttt{ezdxf.render.mesh.MeshTransformer}

Returns geometry as one MeshTransformer object.

Menger Sponge $\text{kind}=0$:

Menger Sponge $\text{kind}=1$:
Menger Sponge kind=2:
Jerusalem Cube $\text{kind}=3$: 
Build a 3D Sierpinsky Pyramid.

class ezdxflib.addons.SierpinskyPyramid(location: Vertex = (0.0, 0.0, 0.0), length: float = 1.0, level: int = 1, sides: int = 4)

Parameters

- `location` – location of base center as (x, y, z) tuple
- `length` – side length
• **level** – subdivide level
• **sides** – sides of base geometry

**render** *(layout: GenericLayoutType, merge: bool = False, dxfattribs: dict = None, matrix: Matrix44 = None, ucs: UCS = None) → None*

Renders the sierpinsky pyramid into layout, set `merge` to `True` for rendering the whole sierpinsky pyramid into one MESH entity, set `merge` to `False` for individual pyramids as MESH entities.

**Parameters**

• **layout** – DXF target layout
• **merge** – *True* for one MESH entity, *False* for individual MESH entities per pyramid
• **dxfattribs** – DXF attributes for the MESH entities
• **matrix** – apply transformation matrix at rendering
• **ucs** – apply UCS at rendering

**pyramids** () → Iterable[ezdxf.render.mesh.MeshTransformer]

Yields all pyramids of the sierpinsky pyramid as individual MeshTransformer objects.

**mesh** () → ezdxf.render.mesh.MeshTransformer

Returns geometry as one MeshTransformer object.

Sierpinsky Pyramid with triangle base:
Sierpinsky Pyramid with square base:
6.11 DXF Internals

- DXF Reference provided by Autodesk.
- DXF Developer Documentation provided by Autodesk.

6.11.1 Basic DXF Structures

DXF File Encoding

DXF R2004 and prior

Drawing files of DXF R2004 (AC1018) and prior are saved as ASCII files with the encoding set by the header variable $DWGCODEPAGE, which is ANSI_1252 by default if $DWGCODEPAGE is not set.

Characters used in the drawing which do not exist in the chosen ASCII encoding are encoded as unicode characters with the schema \U+nnnn. see Unicode table
Known $DWGCODEPAGE encodings

<table>
<thead>
<tr>
<th>DXF</th>
<th>Python</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI_874</td>
<td>cp874</td>
<td>Thai</td>
</tr>
<tr>
<td>ANSI_932</td>
<td>cp932</td>
<td>Japanese</td>
</tr>
<tr>
<td>ANSI_936</td>
<td>gbk</td>
<td>UnifiedChinese</td>
</tr>
<tr>
<td>ANSI_949</td>
<td>cp949</td>
<td>Korean</td>
</tr>
<tr>
<td>ANSI_950</td>
<td>cp950</td>
<td>TradChinese</td>
</tr>
<tr>
<td>ANSI_1250</td>
<td>cp1250</td>
<td>CentralEurope</td>
</tr>
<tr>
<td>ANSI_1251</td>
<td>cp1251</td>
<td>Cyrillic</td>
</tr>
<tr>
<td>ANSI_1252</td>
<td>cp1252</td>
<td>WesternEurope</td>
</tr>
<tr>
<td>ANSI_1253</td>
<td>cp1253</td>
<td>Greek</td>
</tr>
<tr>
<td>ANSI_1254</td>
<td>cp1254</td>
<td>Turkish</td>
</tr>
<tr>
<td>ANSI_1255</td>
<td>cp1255</td>
<td>Hebrew</td>
</tr>
<tr>
<td>ANSI_1256</td>
<td>cp1256</td>
<td>Arabic</td>
</tr>
<tr>
<td>ANSI_1257</td>
<td>cp1257</td>
<td>Baltic</td>
</tr>
<tr>
<td>ANSI_1258</td>
<td>cp1258</td>
<td>Vietnam</td>
</tr>
</tbody>
</table>

DXF R2007 and later

Starting with DXF R2007 (AC1021) the drawing file is UTF-8 encoded, the header variable $DWGCODEPAGE is still in use, but I don’t know, if the setting still has any meaning.

Encoding characters in the unicode schema \U+nnnn is still functional.

See also:

String value encoding

DXF Tags

A Drawing Interchange File is simply an ASCII text file with a file type of .dxf and special formatted text. The basic file structure are DXF tags, a DXF tag consist of a DXF group code as an integer value on its own line and a the DXF value on the following line. In the ezdxf documentation DXF tags will be written as (group code, value).

Group codes are indicating the value type:

<table>
<thead>
<tr>
<th>Group Code</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-9</td>
<td>String (with the introduction of extended symbol names in DXF R2000, the 255-character limit has been increased to 2049 single-byte characters not including the newline at the end of the line)</td>
</tr>
<tr>
<td>10-39</td>
<td>Double precision 3D point value</td>
</tr>
<tr>
<td>40-59</td>
<td>Double-precision floating-point value</td>
</tr>
<tr>
<td>60-79</td>
<td>16-bit integer value</td>
</tr>
<tr>
<td>90-99</td>
<td>32-bit integer value</td>
</tr>
<tr>
<td>100</td>
<td>String (255-character maximum, less for Unicode strings)</td>
</tr>
<tr>
<td>102</td>
<td>String (255-character maximum, less for Unicode strings)</td>
</tr>
<tr>
<td>105</td>
<td>String representing hexadecimal (hex) handle value</td>
</tr>
<tr>
<td>110-119</td>
<td>Double precision floating-point value</td>
</tr>
<tr>
<td>120-129</td>
<td>Double precision floating-point value</td>
</tr>
<tr>
<td>130-139</td>
<td>Double precision floating-point value</td>
</tr>
<tr>
<td>140-149</td>
<td>Double precision scalar floating-point value</td>
</tr>
</tbody>
</table>
## Table 2 – continued from previous page

<table>
<thead>
<tr>
<th>Group Code</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>160-169</td>
<td>64-bit integer value</td>
</tr>
<tr>
<td>170-179</td>
<td>16-bit integer value</td>
</tr>
<tr>
<td>210-239</td>
<td>Double-precision floating-point value</td>
</tr>
<tr>
<td>270-279</td>
<td>16-bit integer value</td>
</tr>
<tr>
<td>280-289</td>
<td>16-bit integer value</td>
</tr>
<tr>
<td>290-299</td>
<td>Boolean flag value</td>
</tr>
<tr>
<td>300-309</td>
<td>Arbitrary text string</td>
</tr>
<tr>
<td>310-319</td>
<td>String representing hex value of binary chunk</td>
</tr>
<tr>
<td>320-329</td>
<td>String representing hex handle value</td>
</tr>
<tr>
<td>330-369</td>
<td>String representing hex object IDs</td>
</tr>
<tr>
<td>370-379</td>
<td>16-bit integer value</td>
</tr>
<tr>
<td>380-389</td>
<td>16-bit integer value</td>
</tr>
<tr>
<td>390-399</td>
<td>String representing hex handle value</td>
</tr>
<tr>
<td>400-409</td>
<td>16-bit integer value</td>
</tr>
<tr>
<td>410-419</td>
<td>String</td>
</tr>
<tr>
<td>420-429</td>
<td>32-bit integer value</td>
</tr>
<tr>
<td>430-439</td>
<td>String</td>
</tr>
<tr>
<td>440-449</td>
<td>32-bit integer value</td>
</tr>
<tr>
<td>450-459</td>
<td>Long</td>
</tr>
<tr>
<td>460-469</td>
<td>Double-precision floating-point value</td>
</tr>
<tr>
<td>470-479</td>
<td>String</td>
</tr>
<tr>
<td>480-481</td>
<td>String representing hex handle value</td>
</tr>
<tr>
<td>999</td>
<td>Comment (string)</td>
</tr>
<tr>
<td>1000-1009</td>
<td>String (same limits as indicated with 0-9 code range)</td>
</tr>
<tr>
<td>1010-1059</td>
<td>Double-precision floating-point value</td>
</tr>
<tr>
<td>1060-1070</td>
<td>16-bit integer value</td>
</tr>
<tr>
<td>1071</td>
<td>32-bit integer value</td>
</tr>
</tbody>
</table>

### Explanation for some important group codes:

<table>
<thead>
<tr>
<th>Group Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DXF structure tag. entity start/end or table entries</td>
</tr>
<tr>
<td>1</td>
<td>The primary text value for an entity</td>
</tr>
<tr>
<td>2</td>
<td>A name: Attribute tag, Block name, and so on. Also used to identify a DXF section or table name.</td>
</tr>
<tr>
<td>3-4</td>
<td>Other textual or name values</td>
</tr>
<tr>
<td>5</td>
<td>Entity handle as hex string (fixed)</td>
</tr>
<tr>
<td>6</td>
<td>Line type name (fixed)</td>
</tr>
<tr>
<td>7</td>
<td>Text style name (fixed)</td>
</tr>
<tr>
<td>8</td>
<td>Layer name (fixed)</td>
</tr>
<tr>
<td>9</td>
<td>Variable name identifier (used only in HEADER section of the DXF file)</td>
</tr>
<tr>
<td>10</td>
<td>Primary X coordinate (start point of a Line or Text entity, center of a Circle, etc.)</td>
</tr>
<tr>
<td>11-18</td>
<td>Other X coordinates</td>
</tr>
<tr>
<td>20</td>
<td>Primary Y coordinate. 2n values always correspond to 1n values and immediately follow them in the file (expected by ezdxf!)</td>
</tr>
<tr>
<td>21-28</td>
<td>Other Y coordinates</td>
</tr>
<tr>
<td>30</td>
<td>Primary Z coordinate. 3n values always correspond to 1n and 2n values and immediately follow them in the file (expected by ezdxf!)</td>
</tr>
<tr>
<td>31-38</td>
<td>Other Z coordinates</td>
</tr>
<tr>
<td>39</td>
<td>This entity’s thickness if nonzero (fixed)</td>
</tr>
<tr>
<td>40-48</td>
<td>Float values (text height, scale factors, etc.)</td>
</tr>
<tr>
<td>49</td>
<td>Repeated value - multiple 49 groups may appear in one entity for variable length tables (such as the dash lengths in the LTYPE table)</td>
</tr>
</tbody>
</table>

---

6.11. DXF Internals
Table 3 – continued from previous page

<table>
<thead>
<tr>
<th>Group Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>50-58</td>
<td>Angles in degree</td>
</tr>
<tr>
<td>62</td>
<td>Color number (fixed)</td>
</tr>
<tr>
<td>66</td>
<td>“Entities follow” flag (fixed), only in INSERT and POLYLINE entities</td>
</tr>
<tr>
<td>67</td>
<td>Identifies whether entity is in modelspace (0) or paperspace (1)</td>
</tr>
<tr>
<td>68</td>
<td>Identifies whether viewport is on but fully off screen, is not active, or is off</td>
</tr>
<tr>
<td>69</td>
<td>Viewport identification number</td>
</tr>
<tr>
<td>70-78</td>
<td>Integer values such as repeat counts, flag bits, or modes</td>
</tr>
<tr>
<td>210, 220, 230</td>
<td>X, Y, and Z components of extrusion direction (fixed)</td>
</tr>
<tr>
<td>310</td>
<td>Proxy entity graphics as binary encoded data</td>
</tr>
<tr>
<td>330</td>
<td>Owner handle as hex string</td>
</tr>
<tr>
<td>347</td>
<td>MATERIAL handle as hex string</td>
</tr>
<tr>
<td>348</td>
<td>VISUALSTYLE handle as hex string</td>
</tr>
<tr>
<td>370</td>
<td>Lineweight in mm times 100 (e.g. 0.13mm = 13).</td>
</tr>
<tr>
<td>390</td>
<td>PLOTSTYLE handle as hex string</td>
</tr>
<tr>
<td>420</td>
<td>True color value as 0x00RRGGBB 24-bit value</td>
</tr>
<tr>
<td>430</td>
<td>Color name as string</td>
</tr>
<tr>
<td>440</td>
<td>Transparency value 0x020000TT 0 = fully transparent / 255 = opaque</td>
</tr>
<tr>
<td>999</td>
<td>Comments</td>
</tr>
</tbody>
</table>

For explanation of all group codes see: DXF Group Codes in Numerical Order Reference provided by Autodesk

**Extended Data**

Extended data (XDATA) is created by AutoLISP or ObjectARX applications but any other application like ezdxr can also define XDATA. If an entity contains extended data, it **follows** the entity’s normal definition data but ends **before** Embedded Objects.

But extended group codes (>=1000) can appear **before** the XDATA section, an example is the BLOCKBASEPOINT-PARAMETER entity in AutoCAD Civil 3D or AutoCAD Map 3D.
<table>
<thead>
<tr>
<th>Group Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Strings in extended data can be up to 255 bytes long (with the 256th byte reserved for the null character)</td>
</tr>
<tr>
<td>1001</td>
<td>(fixed) Registered application name (ASCII string up to 31 bytes long) for XDATA</td>
</tr>
<tr>
<td>1002</td>
<td>(fixed) An extended data control string can be either '{ ' or '}' '. These braces enable applications to organize their data by subdividing the data into lists. Lists can be nested.</td>
</tr>
<tr>
<td>1003</td>
<td>Name of the layer associated with the extended data</td>
</tr>
<tr>
<td>1004</td>
<td>Binary data is organized into variable-length chunks. The maximum length of each chunk is 127 bytes. In ASCII DXF files, binary data is represented as a string of hexadecimal digits, two per binary byte</td>
</tr>
<tr>
<td>1005</td>
<td>Database Handle of entities in the drawing database, see also: About 1005 Group Codes</td>
</tr>
<tr>
<td>1010, 1020, 1030</td>
<td>Three real values, in the order X, Y, Z. They can be used as a point or vector record.</td>
</tr>
<tr>
<td>1011, 1021, 1031</td>
<td>Unlike a simple 3D point, the world space coordinates are moved, scaled, rotated, mirrored, and stretched along with the parent entity to which the extended data belongs.</td>
</tr>
<tr>
<td>1012, 1012, 1022</td>
<td>Also a 3D point that is scaled, rotated, and mirrored along with the parent (but is not moved or stretched)</td>
</tr>
<tr>
<td>1013, 1023, 1033</td>
<td>Also a 3D point that is scaled, rotated, and mirrored along with the parent (but is not moved or stretched)</td>
</tr>
<tr>
<td>1040</td>
<td>A real value</td>
</tr>
<tr>
<td>1041</td>
<td>Distance, a real value that is scaled along with the parent entity</td>
</tr>
<tr>
<td>1042</td>
<td>Scale Factor, also a real value that is scaled along with the parent. The difference between a distance and a scale factor is application-defined</td>
</tr>
<tr>
<td>1070</td>
<td>A 16-bit integer (signed or unsigned)</td>
</tr>
<tr>
<td>1071</td>
<td>A 32-bit signed (long) integer</td>
</tr>
</tbody>
</table>

The (1001, ...) tag indicates the beginning of extended data. In contrast to normal entity data, with extended data the same group code can appear multiple times, and **order is important**.

Extended data is grouped by registered application name. Each registered application group begins with a (1001, APPID) tag, with the application name as APPID string value. Registered application names correspond to APPID symbol table entries.

An application can use as many APPID names as needed. APPID names are permanent, although they can be purged if they aren’t currently used in the drawing. Each APPID name can have **no more than one data group** attached to each entity. Within an application group, the sequence of extended data groups and their meaning is defined by the application.

### String value encoding

String values stored in a DXF file is plain ASCII or UTF-8, AutoCAD also supports CIF (Common Interchange Format) and MIF (Maker Interchange Format) encoding. The UTF-8 format is only supported in DXF R2007 and later.

ezdxf on import converts all strings into Python unicode strings without encoding or decoding CIF/MIF.

String values containing Unicode characters are represented with control character sequences \U+nnnn. (e.g. r'\TEST\U+7F3A\U+4E4F\U+89E3\U+91CA\U+6B63THIS\U+56FE')

To support the DXF unicode encoding ezdxf registers an encoding codec dxf_backslash_replace, defined in ezdxf. lldxf.encoding().
String values can be stored with these dxf group codes:

- 0 - 9
- 100 - 101
- 300 - 309
- 410 - 419
- 430 - 439
- 470 - 479
- 479
- 999 - 1003

**Multi tag text (MTEXT)**

If the text string is less than 250 characters, all characters appear in tag \((1, \ldots)\). If the text string is longer than 250 characters, the string is divided into 250-character chunks, which appear in one or more \((3, \ldots)\) tags. If \((3, \ldots)\) tags are used, the last group is a \((1, \ldots)\) tag and has fewer than 250 characters:

```
3
... TwoHundredAndFifty Characters ....
3
... TwoHundredAndFifty Characters ....
1
less than TwoHundredAndFifty Characters
```

As far I know this is only supported by the MTEXT entity.

See also:

*Dxf File Encoding*

**DXF R13 and later tag structure**

With the introduction of DXF R13 Autodesk added additional group codes and DXF tag structures to the DXF Standard.

**Subclass Markers**

Subclass markers \((100, \text{Subclass Name})\) divides DXF objects into several sections. Group codes can be reused in different sections. A subclass ends with the following subclass marker or at the beginning of xdata or the end of the object. See *Subclass Marker Example* in the DXF Reference.

**Quote about group codes from the DXF reference**

Some group codes that define an entity always appear; others are optional and appear only if their values differ from the defaults.

**Do not** write programs that rely on the order given here. The end of an entity is indicated by the next 0 group, which begins the next entity or indicates the end of the section.

**Note:** Accommodating DXF files from future releases of AutoCAD will be easier if you write your DXF processing program in a table-driven way, ignore undefined group codes, and make no assumptions about
the order of group codes in an entity. With each new AutoCAD release, new group codes will be added to entities to accommodate additional features.

**Usage of group codes in subclasses twice**

Some later entities contain the same group code twice for different purposes, so order in the sense of which one comes first is important. (e.g. ATTDEF group code 280)

**Tag order is sometimes important especially for AutoCAD**

In LWPOLYLINE the order of tags is important, if the *count* tag is not the first tag in the AcDbPolyline subclass, AutoCAD will not close the polyline when the *close* flag is set, by the way other applications like BricsCAD ignores the tag order and renders the polyline always correct.

**Extension Dictionary**

The extension dictionary is an optional sequence that stores the handle of a DICTIONARY object that belongs to the current object, which in turn may contain entries. This facility allows attachment of arbitrary database objects to any database object. Any object or entity may have this section.

The extension dictionary tag sequence:

```plaintext
102
{ACAD_XDICTIONARY
360
Hard-owner ID/handle to owner dictionary
102
}
```

**Persistent Reactors**

Persistent reactors are an optional sequence that stores object handles of objects registering themselves as reactors on the current object. Any object or entity may have this section.

The persistent reactors tag sequence:

```plaintext
102
{ACAD_REACTORS
330
first Soft-pointer ID/handle to owner dictionary
330
second Soft-pointer ID/handle to owner dictionary
...
102
}
```

**Application-Defined Codes**

Starting at DXF R13, DXF objects can contain application-defined codes outside of XDATA. This application-defined codes can contain any tag except (0, ...) and (102, '{...'). "{YOURAPPID" means the APPID string with an preceding "{". The application defined data tag sequence:
(102, 'YOURAPPID') is also a valid closing tag:

```python
102
{YOURAPPID
...
102
}
```

All groups defined with a beginning `(102, ...)` appear in the DXF reference before the first subclass marker, I don’t know if these groups can appear after the first or any subclass marker. ezdxf accepts them at any position, and by default ezdxf adds new app data in front of the first subclass marker to the first tag section of an DXF object.

**Exception XRECORD:** Tags with group code 102 and a value string without a preceding “{” or the scheme “YOURAPPID)”, should be treated as usual group codes.

### Embedded Objects

The concept of embedded objects was introduced with AutoCAD 2018 (DXF version AC1032) and this is the only information I found about it at the Autodesk knowledge base: [Embedded and Encapsulated Objects](https://knowledge.autodesk.com/support/autocad/PTX/PTX Chlorinated Solvents/Help.html)

Quote from Embedded and Encapsulated Objects:

> For DXF filing, the embedded object must be filed out and in after all the data of the encapsulating object has been filed out and in.

> A separator is needed between the encapsulating object’s data and the subsequent embedded object’s data. The separator must be similar in function to the group 0 or 100 in that it must cause the filer to stop reading data. The normal DXF group code 0 cannot be used because DXF proxies use it to determine when to stop reading data. The group code 100 could have been used, but it might have caused confusion when manually reading a DXF file, and there was a need to distinguish when an embedded object is about to be written out in order to do some internal bookkeeping. Therefore, the DXF group code 101 was introduced.

**Hard facts:**

- Embedded object start with `(101, "Embedded Object")` tag
- Embedded object is appended to the encapsulated object
- `(101, "Embedded Object")` tag is the end of the encapsulating object, also of its *Extended Data*
- Embedded object tags can contain any group code except the DXF structure tag (0, ...)

**Unconfirmed assumptions:**

- The encapsulating object can contain more than one embedded object.
- Embedded objects separated by `(101, "Embedded Object")` tags
- every entity can contain embedded objects
- XDATA sections replaced by embedded objects, at least for the MTEXT entity

Real world example from an AutoCAD 2018 file:
100 <<< start of encapsulating object
AcDbMText
10
2762.148
20
2327.073
30
0.0
40
2.5
41
18.852
46
0.0
71
1
72
5
1
{\fArial\b0\i0\c162|p34;CHANGE;\P\P\PTEXT}
73
1
44
1.0
101 <<< start of embedded object
Embedded Object
70
1
10
1.0
20
0.0
30
0.0
11
2762.148
21
2327.073
31
0.0
40
18.852
41
0.0
42
15.428
43
15.043
71
2
72
1
44
18.852
45
12.5
73

(continues on next page)
Handles

A handle is an arbitrary but in your DXF file unique hex value as string like ‘10FF’. It is common to to use uppercase letters for hex numbers. Handle can have up to 16 hexadecimal digits (8 bytes).

For DXF R10 until R12 the usage of handles was optional. The header variable $HANDLING set to 1 indicate the usage of handles, else $HANDLING is 0 or missing.

For DXF R13 and later the usage of handles is mandatory and the header variable $HANDLING was removed.

The $HANDSEED variable in the header section should be greater than the biggest handle used in the DXF file, so a CAD application can assign handle values starting with the $HANDSEED value. But as always, don’t rely on the header variable it could be wrong, AutoCAD ignores this value.

Handle Definition

Entity handle definition is always the (5, ...) except for entities of the DIMSTYLE table (105, ...), because the DIMSTYLE entity has also a group code 5 tag for DIMBLK.

Handle Pointer

A pointer is a reference to a DXF object in the same DXF file. There are four types of pointers:

- Soft-pointer handle
- Hard-pointer handle
- Soft-owner handle
- Hard-owner handle

Also, a group code range for “arbitrary” handles is defined to allow convenient storage of handle values that are unchanged at any operation (AutoCAD).

Pointer and Ownership

A pointer is a reference that indicates usage, but not possession or responsibility, for another object. A pointer reference means that the object uses the other object in some way, and shares access to it. An ownership reference means that an owner object is responsible for the objects for which it has an owner handle. An object can have any number of pointer references associated with it, but it can have only one owner.

Hard and Soft References

Hard references, whether they are pointer or owner, protect an object from being purged. Soft references do not.
In AutoCAD, block definitions and complex entities are hard owners of their elements. A symbol table and dictionaries are soft owners of their elements. Polyline entities are hard owners of their vertex and seqend entities. Insert entities are hard owners of their attrib and seqend entities.

When establishing a reference to another object, it is recommended that you think about whether the reference should protect an object from the PURGE command.

**Arbitrary Handles**

Arbitrary handles are distinct in that they are not translated to session-persistent identifiers internally, or to entity names in AutoLISP, and so on. They are stored as handles. When handle values are translated in drawing-merge operations, arbitrary handles are ignored.

In all environments, arbitrary handles can be exchanged for entity names of the current drawing by means of the handent functions. A common usage of arbitrary handles is to refer to objects in external DXF and DWG files.

**About 1005 Group Codes**

(1005, ...) xdata have the same behavior and semantics as soft pointers, which means that they are translated whenever the host object is merged into a different drawing. However, 1005 items are not translated to session-persistent identifiers or internal entity names in AutoLISP and ObjectARX. They are stored as handles.

**DXF File Structure**

A DXF File is simply an ASCII text file with a file type of .dxf and special formatted text. The basic file structure are DXF tags, a DXF tag consist of a DXF group code as an integer value on its own line and a the DXF value on the following line. In the ezdxf documentation DXF tags will be written as (group code, value). There exist a binary DXF format, but it seems that it is not often used and for reducing file size, zipping is much more efficient. *ezdxf* does support reading binary encoded DXF files.

**See also:**

For more information about DXF tags see: *DXF Tags*

A usual DXF file is organized in sections, starting with the DXF tag (0, ‘SECTION’) and ending with the DXF tag (0, ‘ENDSEC’). The (0, ‘EOF’) tag signals the end of file.

1. **HEADER:** General information about the drawing is found in this section of the DXF file. Each parameter has a variable name starting with ‘$’ and an associated value. Has to be the first section.

2. **CLASSES:** Holds the information for application defined classes. (DXF R13 and later)

3. **TABLES:** Contains several tables for style and property definitions.
   - Linetype table (LTYPE)
   - Layer table (LAYER)
   - Text Style table (STYLE)
   - View table (VIEW): (IMHO) layout of the CAD working space, only interesting for interactive CAD applications
   - Viewport configuration table (VPORT): The VPORT table is unique in that it may contain several entries with the same name (indicating a multiple-viewport configuration). The entries corresponding to the active viewport configuration all have the name *ACTIVE. The first such entry describes the current viewport.
   - Dimension Style table (DIMSTYLE)
• User Coordinate System table (UCS) (IMHO) only interesting for interactive CAD applications

• Application Identification table (APPID): Table of names for all applications registered with a drawing.

• Block Record table (BLOCK_RECORD) (DXF R13 and Later)

4. BLOCKS: Contains all block definitions. The block name *Model_Space or *MODEL_SPACE is reserved for the drawing modelspace and the block name *Paper_Space or *PAPER_SPACE is reserved for the active paperspace layout. Both block definitions are empty, the content of the modelspace and the active paperspace is stored in the ENTITIES section. The entities of other layouts are stored in special block definitions called *Paper_Spacennn, nnn is an arbitrary but unique number.

5. ENTITIES: Contains all graphical entities of the modelspace and the active paperspace layout. Entities of other layouts are stored in the BLOCKS sections.

6. OBJECTS: Contains all non-graphical objects of the drawing (DXF R13 and later)

7. THUMBNAILIMAGE: Contains a preview image of the DXF file, it is optional and can usually be ignored. (DXF R13 and later)

8. ACDS DATA: (DXF R2013 and later) No information in the DXF reference about this section

9. END OF FILE

For further information read the original DXF Reference.

Structure of a usual DXF R12 file:

```
0 <<< Begin HEADER section, has to be the first section
SECTION 2 HEADER
   <<< Header variable items go here
0 <<< End HEADER section
ENDSEC 0 <<< Begin TABLES section
SECTION 2 TABLES 0 TABLE 2 VPORT 70 <<< viewport table maximum item count
   <<< viewport table items go here
0 ENDTAB 0 TABLE 2 APPID, DIMSTYLE, LTYPE, LAYER, STYLE, UCS, VIEW, or VPORT 70 <<< Table maximum item count, a not reliable value and ignored by AutoCAD
   <<< Table items go here
0 ENDTAB 0 <<< End TABLES section
ENDSEC 0 <<< Begin BLOCKS section
```

(continues on next page)
Minimal DXF Content

DXF R12

Contrary to the previous chapter, the DXF R12 format (AC1009) and prior requires just the ENTITIES section:

```
0
SECTION
2
ENTITIES
0
ENDSEC
0
EOF
```

DXF R13/R14 and later

DXF version R13/14 and later needs much more DXF content than DXF R12.

Required sections: HEADER, CLASSES, TABLES, ENTITIES, OBJECTS

The HEADER section requires two entries:

- $ACADVER
- $HANDSEED

The CLASSES section can be empty, but some DXF entities requires class definitions to work in AutoCAD.

The TABLES section requires following tables:

- VPORT entry *ACTIVE is not required! Empty table is ok for AutoCAD.
- LTYPE with at least the following line types defined:
  - BYBLOCK
  - BYLAYER
  - CONTINUOUS
- LAYER with at least an entry for layer ‘0’
- STYLE with at least an entry for style STANDARD
- VIEW can be empty
• UCS can be empty
• APPID with at least an entry for ACAD
• DIMSTYLE with at least an entry for style STANDARD
• BLOCK_RECORDS with two entries:
  – *MODEL_SPACE
  – *PAPER_SPACE
The BLOCKS section requires two BLOCKS:
• *MODEL_SPACE
• *PAPER_SPACE
The ENTITIES section can be empty.

The OBJECTS section requires following entities:
• DICTIONARY - the root dict - one entry named ACAD_GROUP
• DICTIONARY ACAD_GROUP can be empty

Minimal DXF to download: https://github.com/mozman/ezdxf/tree/master/examples_dxf

**Data Model**

**Database Objects**

(from the DXF Reference)

AutoCAD drawings consist largely of structured containers for database objects. Database objects each have the following features:

• A handle whose value is unique to the drawing/DXF file, and is constant for the lifetime of the drawing. This format has existed since AutoCAD Release 10, and as of AutoCAD Release 13, handles are always enabled.
• An optional XDATA table, as entities have had since AutoCAD Release 11.
• An optional persistent reactor table.
• An optional ownership pointer to an extension dictionary which, in turn, owns subobjects placed in it by an application.

Symbol tables and symbol table records are database objects and, thus, have a handle. They can also have xdata and persistent reactors in their DXF records.

**DXF R12 Data Model**

The DXF R12 data model is identical to the file structure:

• HEADER section: common settings for the DXF drawing
• TABLES section: definitions for LAYERS, LINETYPE, STYLES . . .
• BLOCKS section: block definitions and its content
• ENTITIES section: modelspace and paperspace content
References are realized by simple names. The INSERT entity references the BLOCK definition by the BLOCK name, a TEXT entity defines the associated STYLE and LAYER by its name and so on, handles are not needed. Layout association of graphical entities in the ENTITIES section by the paper_space tag (67, 0 or 1), 0 or missing tag means model space, 1 means paperspace. The content of BLOCK definitions is enclosed by the BLOCK and the ENDBLK entity, no additional references are needed.

A clean and simple file structure and data model, which seems to be the reason why the DXF R12 Reference (released 1992) is still a widely used file format and Autodesk/AutoCAD supports the format by reading and writing DXF R12 files until today (DXF R13/R14 has no writing support by AutoCAD!).

**TODO: list of available entities**

See also:

More information about the DXF [DXF File Structure](#)

### DXF R13+ Data Model

With the DXF R13 file format, handles are mandatory and they are really used for organizing the new data structures introduced with DXF R13.

The HEADER section is still the same with just more available settings.

The new CLASSES section contains AutoCAD specific data, has to be written like AutoCAD it does, but must not be understood.

The TABLES section got a new BLOCK_RECORD table - see [Block Management Structures](#) for more information.

The BLOCKS sections is mostly the same, but with handles, owner tags and new ENTITY types. Not active paperspace layouts store their content also in the BLOCKS section - see [Layout Management Structures](#) for more information.

The ENTITIES section is also mostly same, but with handles, owner tags and new ENTITY types.

**TODO: list of new available entities**

And the new OBJECTS section - now it's getting complicated!

Most information about the OBJECTS section is just guessed or gathered by trail and error, because the documentation of the OBJECTS section and its objects in the DXF reference provided by Autodesk is very shallow. This is also the reason why I started the DXF Internals section, may be it helps other developers to start one or two steps above level zero.

The OBJECTS sections stores all the non-graphical entities of the DXF drawing. Non-graphical entities from now on just called ‘DXF objects’ to differentiate them from graphical entities, just called ‘entities’. The OBJECTS section follows commonly the ENTITIES section, but this is not mandatory.

DXF R13 introduces several new DXF objects, which resides exclusive in the OBJECTS section, taken from the DXF R14 reference, because I have no access to the DXF R13 reference, the DXF R13 reference is a compiled .hlp file which can’t be read on Windows 10, a drastic real world example why it is better to avoid closed (proprietary) data formats :):

- **DICTIONARY**: a general structural entity as a <name: handle> container
- **ACDBDICTIONARYWDFLT**: a DICTIONARY with a default value
- **DICTIONARYVAR**: used by AutoCAD to store named values in the database
- **ACAD_PROXY_OBJECT**: proxy object for entities created by other applications than AutoCAD
- **GROUP**: groups graphical entities without the need of a BLOCK definition
- **IDBUFFER**: just a list of references to objects
• IMAGEDEF: IMAGE definition structure, required by the IMAGE entity
• IMAGEDEF_REACTOR: also required by the IMAGE entity
• LAYER_INDEX: container for LAYER names
• MLINESTYLE
• OBJECT_PTR
• RASTERVARIABLES
• SPATIAL_INDEX: is always written out empty to a DXF file. This object can be ignored.
• SPATIAL_FILTER
• SORTENTSTABLE: control for regeneration/redraw order of entities
• XRECORD: used to store and manage arbitrary data. This object is similar in concept to XDATA but is not limited by size or order. Not supported by R13c0 through R13c3.

Still missing the LAYOUT object, which is mandatory in DXF R2000 to manage multiple paperspace layouts. I don’t know how DXF R13/R14 manages multiple layouts or if they even support this feature, but I don’t care much about DXF R13/R14, because AutoCAD has no write support for this two formats anymore. ezdxf tries to upgrade this two DXF versions to DXF R2000 with the advantage of only two different data models to support: DXF R12 and DXF R2000+

New objects introduced by DXF R2000:
• LAYOUT: management object for modelspace and multiple paperspace layouts
• ACDBPLACEHOLDER: surprise - just a place holder

New objects in DXF R2004:
• DIMASSOC
• LAYER_FILTER
• MATERIAL
• PLOTSETTINGS
• VBA_PROJECT

New objects in DXF R2007:
• DATATABLE
• FIELD
• LIGHTLIST
• RENDER
• RENDERENVIRONMENT
• MENTALRAYRENDERSETTINGS
• RENDERGLOBAL
• SECTION
• SUNSTUDY
• TABLESTYLE
• UNDERLAYDEFINITION
• VISUALSTYLE
• WIPEOUTVARIABLES

New objects in DXF R2013:
• GEODATA

New objects in DXF R2018:
• ACDBNAVISWORKSMODELDEF

Undocumented objects:
• SCALE
• ACDBSECTIONVIEWSTYLE
• FIELDLIST

Objects Organisation

Many objects in the OBJECTS section are organized in a tree-like structure of DICTIONARY objects. Starting point for this data structure is the ‘root’ DICTIONARY with several entries to other DICTIONARY objects. The root DICTIONARY has to be the first object in the OBJECTS section. The management dicts for GROUP and LAYOUT objects are really important, but IMHO most of the other management tables are optional and for the most use cases not necessary. The ezdxf template for DXF R2018 contains only these entries in the root dict and most of them pointing to an empty DICTIONARY:

• ACAD_COLOR: points to an empty DICTIONARY
• ACAD_GROUP: supported by ezdxf
• ACAD_LAYOUT: supported by ezdxf
• ACAD_MATERIAL: points to an empty DICTIONARY
• ACAD_MLEADERSTYLE: points to an empty DICTIONARY
• ACAD_MLINESTYLE: points to an empty DICTIONARY
• ACAD_PLOTSETTINGS: points to an empty DICTIONARY
• ACAD_PLOTSTYLENAME: points to ACDBDICTIONARYWDFLT with one entry: ‘Normal’
• ACAD_SCALELIST: points to an empty DICTIONARY
• ACAD_TABLESTYLE: points to an empty DICTIONARY
• ACAD_VISUALSTYLE: points to an empty DICTIONARY

Root DICTIONARY content for DXF R2018

```
0 SECTION
2 << start of the OBJECTS section
OBJECTS
0      << root DICTIONARY has to be the first object in the OBJECTS section
DICTIONARY
5      << handle
C
330    << owner tag
0      << always #0, has no owner
```
100
AcDbDictionary
281  <<< hard owner flag
1
3  <<< first entry
ACAD_CIP_PREVIOUS_PRODUCT_INFO
350  <<< handle to target (pointer)
78B  <<< points to a XRECORD with product info about the creator application
3  <<< entry with unknown meaning, if I should guess: something with about colors
→...
ACAD_COLOR
350
4FB  <<< points to a DICTIONARY
3  <<< entry with unknown meaning
ACAD_DETAILVIEWSTYLE
350
7ED  <<< points to a DICTIONARY
3  <<< GROUP management, mandatory in all DXF versions
ACAD_GROUP
350
4FC  <<< points to a DICTIONARY
3  <<< LAYOUT management, mandatory if more than the *active* paperspace is used
ACAD_LAYOUT
350
4FD  <<< points to a DICTIONARY
3  <<< MATERIAL management
ACAD_MATERIAL
350
4FE  <<< points to a DICTIONARY
3  <<< MLEADERSTYLE management
ACAD_MLEADERSTYLE
350
4FF  <<< points to a DICTIONARY
3  <<< MLINESTYLE management
ACAD_MLINESTYLE
350
500  <<< points to a DICTIONARY
3  <<< PLOTSETTINGS management
ACAD_PLOTSETTINGS
350
501  <<< points to a DICTIONARY
3  <<< plot style name management
ACAD_PLOTSTYLENAME
350
503  <<< points to a ACDBDICTIONARYWDFLT
3  <<< SCALE management
ACAD_SCALELIST
350
504  <<< points to a DICTIONARY
3  <<< entry with unknown meaning
ACAD_SECTIONVIEWSTYLE
350
7EB  <<< points to a DICTIONARY
3  <<< TABLESTYLE management
ACAD_TABLESTYLE
350
505  <<< points to a DICTIONARY
(continued on next page)
6.11.2 DXF Structures

DXF Sections

HEADER Section

In DXF R12 and prior the HEADER section was optional, but since DXF R13 the HEADER section is mandatory. The overall structure is:

```
0  <<< Begin HEADER section
SECTION
2
HEADER
9
$ACADVER  <<< Header variable items go here
1
AC1009
...
0
ENDSEC  <<< End HEADER section
```

A header variable has a name defined by a (9, Name) tag and following value tags.

See also:

Documentation of ezdxf `HeaderSection` class.

DXF Reference: Header Variables

CLASSES Section

The CLASSES section contains CLASS definitions which are only important for Autodesk products, some DXF entities require a class definition or AutoCAD will not open the DXF file.

The CLASSES sections was introduced with DXF AC1015 (AutoCAD Release R13).
See also:

DXF Reference: About the DXF CLASSES Section

Documentation of ezdxf ClassesSection class.

The CLASSES section in DXF files holds the information for application-defined classes whose instances appear in the BLOCKS, ENTITIES, and OBJECTS sections of the database. It is assumed that a class definition is permanently fixed in the class hierarchy. All fields are required.

Update 2019-03-03:

Class names are not unique, Autodesk Architectural Desktop 2007 uses the same name, but with different CPP class names in the CLASS section, so storing classes in a dictionary by name as key caused loss of class entries in ezdxf, using a tuple of (name, cpp_class_name) as storage key solved the problem.

CLASS Entities

See also:

DXF Reference: Group Codes for the CLASS entity

CLASS entities have no handle and therefore ezdxf does not store the CLASS entity in the drawing entities database!

```plaintext
0
SECTION 2 <<< begin CLASSES section
CLASSES 0 <<< first CLASS entity
CLASS 1 <<< class DXF entity name; THIS ENTRY IS MAYBE NOT UNIQUE
ACDBDICTIONARYWDFLT 2 <<< C++ class name; always unique
AcDbDictionaryWithDefault 3 <<< application name
ObjectDBX Classes 90 <<< proxy capabilities flags 0
91 <<< instance counter for custom class, since DXF version AC1018 (R2004) 0 <<< no problem if the counter is wrong, AutoCAD doesn't care about 280 <<< was-a-proxy flag. Set to 1 if class was not loaded when this DXF file was created, and 0 otherwise 0
281 <<< is-an-entity flag. Set to 1 if class reside in the BLOCKS or ENTITIES section. If 0, instances may appear only in the OBJECTS section 0
0 0 <<< second CLASS entity
CLASS ...
...
0 <<< end of CLASSES section
ENDSEC
```

TABLES Section

TODO
**BLOCKS Section**

The BLOCKS section contains all BLOCK definitions, beside the normal reusable BLOCKS used by the INSERT entity, all layouts, as there are the modelspace and all paperspace layouts, have at least a corresponding BLOCK definition in the BLOCKS section. The name of the modelspace BLOCK is “*Model_Space*” (DXF R12: “$MODEL_SPACE”) and the name of the active paperspace BLOCK is “*Paper_Space*” (DXF R12: “$PAPER_SPACE”), the entities of these two layouts are stored in the ENTITIES section, the inactive paperspace layouts are named by the scheme “*Paper_Spacennnn*”, and the content of the inactive paperspace layouts are stored in their BLOCK definition in the BLOCKS section.

The content entities of blocks are stored between the BLOCK and the ENDBLK entity.

**BLOCKS section structure:**

```
0 <<< start of a SECTION
SECTION
2 <<< start of BLOCKS section
BLOCKS
0 <<< start of 1. BLOCK definition
BLOCK
... <<< Block content
...
0 <<< end of 1. Block definition
ENDBLK
0 <<< start of 2. BLOCK definition
BLOCK
... <<< Block content
...
0 <<< end of 2. Block definition
ENDBLK
0 <<< end of BLOCKS section
ENDSEC
```

See also:

*Block Management Structures*  *Layout Management Structures*

**ENTITIES Section**

TODO

**OBJECTS Section**

Objects in the OBJECTS section are organized in a hierarchical tree order, starting with the named objects dictionary as the first entity in the OBJECTS section (Drawing.rootdict).

Not all entities in the OBJECTS section are included in this tree, *Extension Dictionary* and XRECORD data of graphical entities are also stored in the OBJECTS section.

**DXF Tables**
VIEW Table

The `VIEW` entry stores a named view of the model or a paperspace layout. This stored views makes parts of the drawing or some view points of the model in a CAD applications more accessible. This views have no influence to the drawing content or to the generated output by exporting PDFs or plotting on paper sheets, they are just for the convenience of CAD application users.

Using `ezdxf` you have access to the views table by the attribute `Drawing.views`. The views table itself is not stored in the entity database, but the table entries are stored in entity database, and can be accessed by its handle.

**DXF R12**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VIEW</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>&lt;&lt;&lt; name of view</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VIEWNAME</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>&lt;&lt;&lt; flags bit-coded: 1st bit -&gt; (0/1 = modelspace/paperspace)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>&lt;&lt;&lt; modelspace</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>&lt;&lt;&lt; view width in Display Coordinate System (DCS)</td>
<td>20.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>&lt;&lt;&lt; view center point in DCS</td>
<td>40.36</td>
<td>&lt;&lt;&lt; x value</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>&lt;&lt;&lt; group code for y value</td>
<td>15.86</td>
<td>&lt;&lt;&lt; y value</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>&lt;&lt;&lt; view height in DCS</td>
<td>17.91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>&lt;&lt;&lt; view direction from target point, 3D vector</td>
<td>0.0</td>
<td>&lt;&lt;&lt; x value</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>&lt;&lt;&lt; group code for y value</td>
<td>0.0</td>
<td>&lt;&lt;&lt; y value</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>&lt;&lt;&lt; group code for z value</td>
<td>1.0</td>
<td>&lt;&lt;&lt; z value</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>&lt;&lt;&lt; target point in WCS</td>
<td>0.0</td>
<td>&lt;&lt;&lt; x value</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>&lt;&lt;&lt; group code for y value</td>
<td>0.0</td>
<td>&lt;&lt;&lt; y value</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>&lt;&lt;&lt; group code for z value</td>
<td>0.0</td>
<td>&lt;&lt;&lt; z value</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>&lt;&lt;&lt; lens (focal) length</td>
<td>50.0</td>
<td>&lt;&lt;&lt; 50mm</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>&lt;&lt;&lt; front clipping plane, offset from target</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>&lt;&lt;&lt; back clipping plane, offset from target</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>&lt;&lt;&lt; twist angle</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>&lt;&lt;&lt; view mode</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See also:

*Coordinate Systems*
DXF R2000+

Mostly the same structure as DXF R12, but with handle, owner tag and subclass markers.

```plaintext
0 <<< adding the VIEW table head, just for information
TABLE
2 <<< table name
VIEW
5 <<< handle of table, see owner tag of VIEW table entry
37C
330 <<< owner tag of table, always #0
0
100 <<< subclass marker
AcDbSymbolTable
70 <<< VIEW table (max.) count, not reliable (ignore)
9
0 <<< first VIEW table entry
VIEW
5 <<< handle
3EA
330 <<< owner, the VIEW table is the owner of the VIEW entry
37C <<< handle of the VIEW table
100 <<< subclass marker
AcDbSymbolTableRecord
100 <<< subclass marker
AcDbViewTableRecord
2 <<< view name, from here all the same as DXF R12
VIEWNAME
70
0
40
20.01
10
40.36
20
15.86
41
17.91
11
0.0
21
0.0
31
1.0
12
0.0
22
0.0
32
0.0
42
50.0
43
0.0
44
0.0
50
(continues on next page)
```
DXF R2000+ supports additional features in the VIEW entry, see the VIEW table reference provided by Autodesk.

**VPORT Configuration Table**

The VPORT table stores the modelspace viewport configurations. A viewport configuration is a tiled view of multiple viewports or just one viewport.

In contrast to other tables the VPORT table can have multiple entries with the same name, because all VPORT entries of a multi-viewport configuration are having the same name - the viewport configuration name. The name of the actual displayed viewport configuration is '*ACTIVE*', as always table entry names are case insensitive ('*ACTIVE*' == '*Active*').

The available display area in AutoCAD has normalized coordinates, the lower-left corner is (0, 0) and the upper-right corner is (1, 1) regardless of the true aspect ratio and available display area in pixels. A single viewport configuration has one VPORT entry '*ACTIVE*' with the lower-left corner (0, 0) and the upper-right corner (1, 1).

The following statements refer to a 2D plan view: the view-target-point defines the origin of the DCS (Display Coordinate system), the view-direction vector defines the z-axis of the DCS, the view-center-point (in DCS) defines the point in modelspace translated to the center point of the viewport, the view height and the aspect-ratio defines how much of the modelspace is displayed. AutoCAD tries to fit the modelspace area into the available viewport space e.g.
view height is 15 units and aspect-ratio is 2.0 the modelspace to display is 30 units wide and 15 units high, if the viewport has an aspect ratio of 1.0, AutoCAD displays 30x30 units of the modelspace in the viewport. If the modelspace aspect-ratio is 1.0 the modelspace to display is 15x15 units and fits properly into the viewport area.

But tests show that the translation of the view-center-point to the middle of the viewport not always work as I expected. (still digging...)

Note: All floating point values are rounded to 2 decimal places for better readability.

**DXF R12**

Multi-viewport configuration with three viewports.

```
0  <<< table start
TABLE
2  <<< table type
VPORT
70  <<< VPORT table (max.) count, not reliable (ignore)
3
0  <<< first VPORT entry
VPORT
2  <<< VPORT (configuration) name
*ACTIVE
70  <<< standard flags, bit-coded
0
10  <<< lower-left corner of viewport
0.45  <<< x value, virtual coordinates in range [0 - 1]
20  <<< group code for y value
0.0  <<< y value, virtual coordinates in range [0 - 1]
11  <<< upper-right corner of viewport
1.0  <<< x value, virtual coordinates in range [0 - 1]
21  <<< group code for y value
1.0  <<< y value, virtual coordinates in range [0 - 1]
12  <<< view center point (in DCS), ???
13.71  <<< x value
22  <<< group code for y value
0.02  <<< y value
13  <<< snap base point (in DCS)
0.0  <<< x value
23  <<< group code for y value
0.0  <<< y value
14  <<< snap spacing X and Y
1.0  <<< x value
24  <<< group code for y value
1.0  <<< y value
15  <<< grid spacing X and Y
0.0  <<< x value
25  <<< group code for y value
0.0  <<< y value
16  <<< view direction from target point (in WCS), defines the z-axis of the DCS
1.0  <<< x value
26  <<< group code for y value
-1.0  <<< y value
36  <<< group code for z value
1.0  <<< z value
```
17 <<< view target point (in WCS), defines the origin of the DCS
0.0 <<< x value
27 <<< group code for y value
0.0 <<< y value
37 <<< group code for z value
0.0 <<< z value
40 <<< view height
35.22
41 <<< viewport aspect ratio
0.99
42 <<< lens (focal) length
50.0 <<< 50mm
43 <<< front clipping planes, offsets from target point
0.0
44 <<< back clipping planes, offsets from target point
0.0
50 <<< snap rotation angle
0.0
51 <<< view twist angle
0.0
71 <<< view mode
0
72 <<< circle zoom percent
1000
73 <<< fast zoom setting
1
74 <<< UCSICON setting
3
75 <<< snap on/off
0
76 <<< grid on/off
0
77 <<< snap style
0
78 <<< snap isopair
0
0 <<< next VPORT entry
VPORT
2 <<< VPORT (configuration) name
*ACTIVE <<< same as first VPORT entry
70
0
10
0.0
20
0.5
11
0.45
21
1.0
12
8.21
22
9.41
...
0 <<< next VPORT entry
(continues on next page)
VPOR
2  <<< VPORT (configuration) name
*ACTIVE <<< same as first VPORT entry
70
0
10
0.0
20
0.0
11
0.45
21
0.5
12
2.01
22
-9.33
...
...
0
ENDTAB

**DXF R2000+**

Mostly the same structure as DXF R12, but with handle, owner tag and subclass markers.

<table>
<thead>
<tr>
<th>0</th>
<th>&lt;&lt;&lt; table start</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>&lt;&lt;&lt; table type</td>
</tr>
</tbody>
</table>
| VPOR
5  | <<< table handle |
151F
330 | <<< owner, table has no owner - always #0 |
0
100 | <<< subclass marker |
AcDbSymbolTable
70 | <<< VPORT table (max.) count, not reliable (ignore) |
3
0 | <<< first VPORT entry |
VPOR
5  | <<< entry handle |
158B
330 | <<< owner, VPORT table is owner of VPORT entry |
151F
100 | <<< subclass marker |
AcDbSymbolTableRecord
100 | <<< subclass marker |
AcDbViewportTableRecord
2  | <<< VPORT (configuration) name |
*ACTIVE
70 | <<< standard flags, bit-coded |
0
10 | <<< lower-left corner of viewport |
0.45 | <<< x value, virtual coordinates in range [0 - 1] |
20 | <<< group code for y value |

(continues on next page)
0.0 <<< y value, virtual coordinates in range [0 - 1]
11 <<< upper-right corner of viewport
1.0 <<< x value, virtual coordinates in range [0 - 1]
21 <<< group code for y value
1.0 <<< y value, virtual coordinates in range [0 - 1]
12 <<< view center point (in DCS)
13.71 <<< x value
22 <<< group code for y value
0.38 <<< y value
13 <<< snap base point (in DCS)
0.0 <<< x value
23 <<< group code for y value
0.0 <<< y value
14 <<< snap spacing X and Y
1.0 <<< x value
24 <<< group code for y value
1.0 <<< y value
15 <<< grid spacing X and Y
0.0 <<< x value
25 <<< group code for y value
0.0 <<< y value
16 <<< view direction from target point (in WCS)
1.0 <<< x value
26 <<< group code for y value
-1.0 <<< y value
36 <<< group code for z value
1.0 <<< z value
17 <<< view target point (in WCS)
0.0 <<< x value
27 <<< group code for y value
0.0 <<< y value
37 <<< group code for z value
0.0 <<< z value
40 <<< view height
35.22
41 <<< viewport aspect ratio
0.99
42 <<< lens (focal) length
50.0 <<< 50mm
43 <<< front clipping planes, offsets from target point
0.0
44 <<< back clipping planes, offsets from target point
0.0
50 <<< snap rotation angle
0.0
51 <<< view twist angle
0.0
71 <<< view mode
0
72 <<< circle zoom percent
1000
73 <<< fast zoom setting
1
74 <<< UCSICON setting
3
75 <<< snap on/off
0
76 <<< grid on/off
77 <<< snap style
78 <<< snap isopair
281 <<< render mode 1-6 (... too many options)
65 <<< Value of UCSVP for this viewport. (0 = UCS will not change when this viewpoint is activated)
1 <<< 1 = then viewport stores its own UCS which will become the current UCS whenever the viewport is activated.
110 <<< UCS origin (3D point)
0.0 <<< x value
120 <<< group code for y value
0.0 <<< y value
130 <<< group code for z value
0.0 <<< z value
111 <<< UCS X-axis (3D vector)
1.0 <<< x value
121 <<< group code for y value
0.0 <<< y value
131 <<< group code for z value
0.0 <<< z value
112 <<< UCS Y-axis (3D vector)
0.0 <<< x value
122 <<< group code for y value
1.0 <<< y value
132 <<< group code for z value
0.0 <<< z value
79 <<< Orthographic type of UCS 0-6 (... too many options)
0 <<< 0 = UCS is not orthographic
146 <<< elevation
0.0
1001 <<< extended data - undocumented
ACAD_NAV_VCDISPLAY
1070
3
0 <<< next VPORT entry
VPORT
5
158C
330
151F
100
AcDbSymbolTableRecord
100
AcDbViewportTableRecord
2 <<< VPORT (configuration) name
*ACTIVE <<< same as first VPORT entry
70
0
10
0.0
20
0.5
11

(continues on next page)
The **LTYPE** table stores all line type definitions of a DXF drawing. Every line type used in the drawing has to have a table entry, or the DXF drawing is invalid for AutoCAD.

DXF R12 supports just simple line types, DXF R2000+ supports also complex line types with text or shapes included. You have access to the line types table by the attribute `Drawing.linetypes`. The line type table itself is not stored in the entity database, but the table entries are stored in entity database, and can be accessed by its handle.

See also:

- DXF Reference: TABLES Section
- DXF Reference: LTYPE Table
Table Structure DXF R12

```
0   <<< start of table
TABLE
2   <<< set table type
LTYPE
70  <<< count of line types defined in this table, AutoCAD ignores this value
9
0   <<< 1. LTYPE table entry
LTYPE
   <<< LTYPE data tags
0
LTYPE
   <<< LTYPE data tags and so on
0
ENDTAB
```

Table Structure DXF R2000+

```
0   <<< start of table
TABLE
2   <<< set table type
LTYPE
5   <<< LTYPE table handle
5F
330  <<< owner tag, tables has no owner
0
100  <<< subclass marker
AcDbSymbolTable
70  <<< count of line types defined in this table, AutoCAD ignores this value
9
0   <<< 1. LTYPE table entry
LTYPE
   <<< LTYPE data tags
0
LTYPE
   <<< LTYPE data tags and so on
0
ENDTAB
```

Simple Line Type

```
ezdf setup for line type ‘CENTER’:

```
### Simple Line Type Tag Structure DXF R2000+

```
0   <<< line type table entry
LTYPE
5   <<< handle of line type
1B1
330 <<< owner handle, handle of LTYPE table
5F
100 <<< subclass marker
AcDbSymbolTableRecord
100 <<< subclass marker
AcDbLinetypeTableRecord
2   <<< line type name
CENTER
70   <<< flags
0
3
Center _____ _____ _____ _____ _____ _____
72
65
73
4
40
2.0
49
1.25
74
0
49
-0.25
74
0
49
0.25
74
0
49
-0.25
74
0
```

### Complex Line Type TEXT

**ezdxf** setup for line type ‘GASLEITUNG’:

```python
dwg.linetypes.new('GASLEITUNG', dxfattribs={
    'description': 'Gasleitung2 ----GAS----GAS----GAS----GAS----GAS----GAS--',
    'length': 1,
    'pattern': 'A,.5, -.2, \"GAS\", \"STANDARD\", S=.1, U=0.0, X=-0.1, Y=-0.05, -.25',
})
```
TEXT Tag Structure

<table>
<thead>
<tr>
<th>0</th>
<th>LTYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>614</td>
</tr>
<tr>
<td>330</td>
<td>5F</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>AcDbSymbolTableRecord</td>
<td>AcDbLinetypeTableRecord</td>
</tr>
<tr>
<td>2</td>
<td>GASLEITUNG</td>
</tr>
<tr>
<td>70</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Gasleitung2 ----GAS----GAS----GAS----GAS----GAS----GAS--</td>
</tr>
<tr>
<td>72</td>
<td>65</td>
</tr>
<tr>
<td>73</td>
<td>3</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
</tr>
<tr>
<td>49</td>
<td>0.5</td>
</tr>
<tr>
<td>74</td>
<td>0</td>
</tr>
<tr>
<td>49</td>
<td>-0.2</td>
</tr>
<tr>
<td>74</td>
<td>2</td>
</tr>
<tr>
<td>75</td>
<td>0</td>
</tr>
<tr>
<td>340</td>
<td>11</td>
</tr>
<tr>
<td>46</td>
<td>0.1</td>
</tr>
<tr>
<td>50</td>
<td>0.0</td>
</tr>
<tr>
<td>44</td>
<td>-0.1</td>
</tr>
<tr>
<td>45</td>
<td>-0.05</td>
</tr>
<tr>
<td>9</td>
<td>GAS</td>
</tr>
<tr>
<td>49</td>
<td>-0.25</td>
</tr>
<tr>
<td>74</td>
<td>0</td>
</tr>
</tbody>
</table>

**Complex Line Type SHAPE**

ezdxf setup for line type ‘GRENZE2’:
```
dwg.linetypes.new('GRENZE2', dxfattribs={
    'description': 'Grenze eckig ----[]-----[]----[]-----[]----[]--',
    'length': 1.45,
    'pattern': 'A,.25,-.1,[132,ltypeshp.shx,x=-.1,s=.1],-.1,1',
})
```

SHAPE Tag Structure

```
0
LTYPE
5
615
330
5F
100        <<< subclass marker
AcDbSymbolTableRecord
100        <<< subclass marker
AcDbLinetypeTableRecord
2
GRENZE2
70
0
3
Grenze eckig ----[]-----[]----[]-----[]----[]--
72
65
73
4
40
1.45
49
0.25
74
0
49
-0.1
74
4
75
132
340
616
46
0.1
50
0.0
44
-0.1
45
0.0
49
-0.1
74
0
49
```

(continues on next page)
1.0
74
0

**DIMSTYLE Table**

The **DIMSTYLE** table stores all dimension style definitions of a DXF drawing.

You have access to the dimension styles table by the attribute `Drawing.dimstyles`.

See also:

- DXF Reference: TABLES Section
- DXF Reference: DIMSTYLE Table

**Table Structure DXF R12**

```
0  <<< start of table
TABLE 2  <<< set table type
DIMSTYLE 70  <<< count of line types defined in this table, AutoCAD ignores this value
9 0  <<< 1. DIMSTYLE table entry
DIMSTYLE 0  <<< DIMSTYLE data tags
DIMSTYLE 0  <<< 2. DIMSTYLE table entry
DIMSTYLE 0  <<< DIMSTYLE data tags and so on
0  <<< end of DIMSTYLE table
ENDTAB
```

**DIMSTYLE Entry DXF R12**

**DIMSTYLE Variables DXF R12**

Source: CADManager Blog

<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIMALT</td>
<td>170</td>
<td>Controls the display of alternate units in dimensions.</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIMALTD</td>
<td>171</td>
<td>Controls the number of decimal places in alternate units. If DIMALT is turned on, DIMALTD sets the number of digits displayed to the right of the decimal point in the alternate measurement.</td>
</tr>
<tr>
<td>DIMALTF</td>
<td>143</td>
<td>Controls the multiplier for alternate units. If DIMALT is turned on, DIMALTF multiplies linear dimensions by a factor to produce a value in an alternate system of measurement. The initial value represents the number of millimeters in an inch.</td>
</tr>
<tr>
<td>DIMAPOST</td>
<td>4</td>
<td>Specifies a text prefix or suffix (or both) to the alternate dimension measurement for all types of dimensions except angular. For instance, if the current units are Architectural, DIMALT is on, DIMALTF is 25.4 (the number of millimeters per inch), DIMALTD is 2, and DIMAPOST is set to “mm”, a distance of 10 units would be displayed as 10&quot;[254.00mm].</td>
</tr>
<tr>
<td>DIMASZ</td>
<td>41</td>
<td>Controls the size of dimension line and leader line arrowheads. Also controls the size of hook lines. Multiples of the arrowhead size determine whether dimension lines and text should fit between the extension lines. DIMASZ is also used to scale arrowhead blocks if set by DIMBLK. DIMASZ has no effect when DIMTSZ is other than zero.</td>
</tr>
<tr>
<td>DIMBLK</td>
<td>5</td>
<td>Sets the arrowhead block displayed at the ends of dimension lines.</td>
</tr>
<tr>
<td>DIMBLK1</td>
<td>6</td>
<td>Sets the arrowhead for the first end of the dimension line when DIMSAH is 1.</td>
</tr>
<tr>
<td>DIMBLK2</td>
<td>7</td>
<td>Sets the arrowhead for the second end of the dimension line when DIMSAH is 1.</td>
</tr>
</tbody>
</table>
### Table 4 – continued from previous page

<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
</table>
| DIMCEN   | 141  | Controls drawing of circle or arc center marks and centerlines by the DIMCENTER, DIMDIAMETER, and DIMRADIUS commands. For DIMDIAMETER and DIMRADIUS, the center mark is drawn only if you place the dimension line outside the circle or arc.  
  • 0 = No center marks or lines are drawn  
  • <0 = Centerlines are drawn  
  • >0 = Center marks are drawn |
| DIMCLRD  | 176  | Assigns colors to dimension lines, arrowheads, and dimension leader lines.  
  • 0 = BYBLOCK  
  • 1-255 = ACI AutoCAD Color Index  
  • 256 = BYLAYER |
| DIMCLRE  | 177  | Assigns colors to dimension extension lines, values like DIMCLRD |
| DIMCLRT  | 178  | Assigns colors to dimension text, values like DIMCLRD |
| DIMDLE   | 46   | Sets the distance the dimension line extends beyond the extension line when oblique strokes are drawn instead of arrowheads. |
| DIMDLI   | 43   | Controls the spacing of the dimension lines in baseline dimensions. Each dimension line is offset from the previous one by this amount, if necessary, to avoid drawing over it. Changes made with DIMDLI are not applied to existing dimensions. |
| DIMEXE   | 44   | Specifies how far to extend the extension line beyond the dimension line. |
| DIMEXO   | 42   | Specifies how far extension lines are offset from origin points. With fixed-length extension lines, this value determines the minimum offset. |

Continued on next page
**Table 4 – continued from previous page**

<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIMGAP</td>
<td>147</td>
<td>Sets the distance around the dimension text when the dimension line breaks to accommodate dimension text. Also sets the gap between annotation and a hook line created with the LEADER command. If you enter a negative value, DIMGAP places a box around the dimension text. DIMGAP is also used as the minimum length for pieces of the dimension line. When the default position for the dimension text is calculated, text is positioned inside the extension lines only if doing so breaks the dimension lines into two segments at least as long as DIMGAP. Text placed above or below the dimension line is moved inside only if there is room for the arrowheads, dimension text, and a margin between them at least as large as DIMGAP: 2 * (DIMASZ + DIMGAP).</td>
</tr>
<tr>
<td>DMLFAC</td>
<td>144</td>
<td>Sets a scale factor for linear dimension measurements. All linear dimension distances, including radii, diameters, and coordinates, are multiplied by DMLFAC before being converted to dimension text. Positive values of DMLFAC are applied to dimensions in both modelspace and paperspace; negative values are applied to paperspace only. DMLFAC applies primarily to nonassociative dimensions (DIMASSOC set 0 or 1). For nonassociative dimensions in paperspace, DMLFAC must be set individually for each layout viewport to accommodate viewport scaling. DMLFAC has no effect on angular dimensions, and is not applied to the values held in DMRND, DMTM, or DMTP.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 4 – continued from previous page

<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
</table>
| DIMLIM  | 72   | Generates dimension limits as the default text. Setting DIMLIM to On turns DIMTOL off.  
|         |      | • 0 = Dimension limits are not generated as default text  
|         |      | • 1 = Dimension limits are generated as default text |
| DIMPOST | 3    | Specifies a text prefix or suffix (or both) to the dimension measurement.  
|         |      | For example, to establish a suffix for millimeters, set DIMPOST to mm; a distance of 19.2 units would be displayed as 19.2 mm. If tolerances are turned on, the suffix is applied to the tolerances as well as to the main dimension.  
|         |      | Use “<>” to indicate placement of the text in relation to the dimension value. For example, enter “<>mm” to display a 5.0 millimeter radial dimension as “5.0mm”. If you entered mm “<>”, the dimension would be displayed as “mm 5.0”. |
| DIMRND  | 45   | Rounds all dimensioning distances to the specified value.  
|         |      | For instance, if DIMRND is set to 0.25, all distances round to the nearest 0.25 unit. If you set DIMRND to 1.0, all distances round to the nearest integer. Note that the number of digits edited after the decimal point depends on the precision set by DIMDEC. DIMRND does not apply to angular dimensions. |
| DIMSAH  | 173  | Controls the display of dimension line arrowhead blocks.  
|         |      | • 0 = Use arrowhead blocks set by DIMBLK  
|         |      | • 1 = Use arrowhead blocks set by DIMBLK1 and DIMBLK2 |

Continued on next page
Table 4 – continued from previous page

<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIMSCALE</td>
<td>40</td>
<td>Sets the overall scale factor applied to dimensioning variables that specify sizes, distances, or offsets. Also affects the leader objects with the LEADER command. Use MLEADERSCALE to scale multileader objects created with the MLEADER command.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0.0 = A reasonable default value is computed based on the scaling between the current model space viewport and paperspace. If you are in paperspace or modelspace and not using the paperspace feature, the scale factor is 1.0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• &gt;0 = A scale factor is computed that leads text sizes, arrowhead sizes, and other scaled distances to plot at their face values.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DIMSCALE does not affect measured lengths, coordinates, or angles. Use DIMSCALE to control the overall scale of dimensions. However, if the current dimension style is annotative, DIMSCALE is automatically set to zero and the dimension scale is controlled by the CANNOSCALE system variable. DIMSCALE cannot be set to a non-zero value when using annotative dimensions.</td>
</tr>
<tr>
<td>DIMSE1</td>
<td>75</td>
<td>Suppresses display of the first extension line.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 = Extension line is not suppressed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1 = Extension line is suppressed</td>
</tr>
<tr>
<td>DIMSE2</td>
<td>76</td>
<td>Suppresses display of the second extension line.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 = Extension line is not suppressed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1 = Extension line is suppressed</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIMSOXD</td>
<td>175</td>
<td>Suppresses arrowheads if not enough space is available inside the extension lines.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 = Arrowheads are not suppressed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1 = Arrowheads are suppressed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If not enough space is available inside the extension lines and DIMTIX is on, setting DIMSOXD to On suppresses the arrowheads. If DIMTIX is off, DIMSOXD has no effect.</td>
</tr>
<tr>
<td>DIMITAD</td>
<td>77</td>
<td>Controls the vertical position of text in relation to the dimension line.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 = Centers the dimension text between the extension lines.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1 = Places the dimension text above the dimension line except when the dimension line is not horizontal and text inside the extension lines is forced horizontal (DIMTIH = 1). The distance from the dimension line to the baseline of the lowest line of text is the current DIMGAP value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 2 = Places the dimension text on the side of the dimension line farthest away from the defining points.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 3 = Places the dimension text to conform to Japanese Industrial Standards (JIS).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 4 = Places the dimension text below the dimension line.</td>
</tr>
<tr>
<td>DIMITFAC</td>
<td>146</td>
<td>Specifies a scale factor for the text height of fractions and tolerance values relative to the dimension text height, as set by DIMITXT.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For example, if DIMITFAC is set to 1.0, the text height of fractions and tolerances is the same height as the dimension text. If DIMITFAC is set to 0.7500, the text height of fractions and tolerances is three-quarters the size of dimension text.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 4 – continued from previous page

<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
</table>
| DIMTlh | 73   | Controls the position of dimension text inside the extension lines for all dimension types except Ordinate.  
- 0 = Aligns text with the dimension line  
- 1 = Draws text horizontally |
| DIMTIX | 174  | Draws text between extension lines.  
- 0 = Varies with the type of dimension. For linear and angular dimensions, text is placed inside the extension lines if there is sufficient room. For radius and diameter dimensions that don’t fit inside the circle or arc, DIMTIX has no effect and always forces the text outside the circle or arc.  
- 1 = Draws dimension text between the extension lines even if it would ordinarily be placed outside those lines |
| DIMTM | 48   | Sets the minimum (or lower) tolerance limit for dimension text when DIMTOL or DIMLIM is on. DIMTM accepts signed values. If DIMTOL is on and DIMTP and DIMTM are set to the same value, a tolerance value is drawn. If DIMTM and DIMTP values differ, the upper tolerance is drawn above the lower, and a plus sign is added to the DIMTP value if it is positive. For DIMTM, the program uses the negative of the value you enter (adding a minus sign if you specify a positive number and a plus sign if you specify a negative number). |

Continued on next page
<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIMTOFL</td>
<td>172</td>
<td>Controls whether a dimension line is drawn between the extension lines even when the text is placed outside. For radius and diameter dimensions (when DIMTIX is off), draws a dimension line inside the circle or arc and places the text, arrowheads, and leader outside.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 = Does not draw dimension lines between the measured points when arrowheads are placed outside the measured points</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1 = Draws dimension lines between the measured points even when arrowheads are placed outside the measured points</td>
</tr>
<tr>
<td>DIMTOH</td>
<td>74</td>
<td>Controls the position of dimension text outside the extension lines.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 = Aligns text with the dimension line</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1 = Draws text horizontally</td>
</tr>
<tr>
<td>DIMTOL</td>
<td>71</td>
<td>Appends tolerances to dimension text. Setting DIMTOL to on turns DIMLIM off.</td>
</tr>
<tr>
<td>DIMTP</td>
<td>47</td>
<td>Sets the maximum (or upper) tolerance limit for dimension text when DIMTOL or DIMLIM is on. DIMTP accepts signed values. If DIMTOL is on and DIMTP and DIMTM are set to the same value, a tolerance value is drawn. If DIMTM and DIMTP values differ, the upper tolerance is drawn above the lower and a plus sign is added to the DIMTP value if it is positive.</td>
</tr>
<tr>
<td>DIMTSZ</td>
<td>142</td>
<td>Specifies the size of oblique strokes drawn instead of arrowheads for linear, radius, and diameter dimensioning.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 = Draws arrowheads.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• &gt;0 = Draws oblique strokes instead of arrowheads. The size of the oblique strokes is determined by this value multiplied by the DIMSCALE value</td>
</tr>
</tbody>
</table>

Continued on next page
Table 4 – continued from previous page

<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIMTVP</td>
<td>145</td>
<td>Controls the vertical position of dimension text above or below the dimension line. The DIMTVP value is used when DIMTAD = 0. The magnitude of the vertical offset of text is the product of the text height and DIMTVP. Setting DIMTVP to 1.0 is equivalent to setting DIMTAD = 1. The dimension line splits to accommodate the text only if the absolute value of DIMTVP is less than 0.7.</td>
</tr>
<tr>
<td>DIMTXT</td>
<td>140</td>
<td>Specifies the height of dimension text, unless the current text style has a fixed height.</td>
</tr>
</tbody>
</table>
| DIMZIN  | 78   | Controls the suppression of zeros in the primary unit value. Values 0-3 affect feet-and-inch dimensions only:  
- 0 = Suppresses zero feet and precisely zero inches  
- 1 = Includes zero feet and precisely zero inches  
- 2 = Includes zero feet and suppresses zero inches  
- 3 = Includes zero inches and suppresses zero feet  
- 4 (Bit 3) = Suppresses leading zeros in decimal dimensions (for example, 0.5000 becomes .5000)  
- 8 (Bit 4) = Suppresses trailing zeros in decimal dimensions (for example, 12.5000 becomes 12.5)  
- 12 (Bit 3+4) = Suppresses both leading and trailing zeros (for example, 0.5000 becomes .5) |

Table Structure DXF R2000+

```
0 <<< start of table
TABLE
2 <<< set table type
DIMSTYLE
5 <<< DIMSTYLE table handle
SF
330 <<< owner tag, tables has no owner
0
```
Additional DIMSTYLE Variables DXF R13/14

Source: CADDManager Blog
<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIMADEC</td>
<td>179</td>
<td>Controls the number of precision places displayed in angular dimensions.</td>
</tr>
<tr>
<td>DIMALTTD</td>
<td>274</td>
<td>Sets the number of decimal places for the tolerance values in the alternate units of a dimension.</td>
</tr>
<tr>
<td>DIMALTTZ</td>
<td>286</td>
<td>Controls suppression of zeros in tolerance values.</td>
</tr>
<tr>
<td>DIMALTU</td>
<td>273</td>
<td>Sets the units format for alternate units of all dimension substyles except Angular.</td>
</tr>
<tr>
<td>DIMALTZ</td>
<td>285</td>
<td>Controls the suppression of zeros for alternate unit dimension values. DIMALTZ values 0-3 affect feet-and-inch dimensions only.</td>
</tr>
<tr>
<td>DIMAUNIT</td>
<td>275</td>
<td>Sets the units format for angular dimensions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 = Decimal degrees</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1 = Degrees/minutes/seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 2 = Grad</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 3 = Radians</td>
</tr>
<tr>
<td>DIMBLK_HANDLE</td>
<td>342</td>
<td>defines DIMBLK as handle to the BLOCK RECORD entry</td>
</tr>
<tr>
<td>DIMBLK1_HANDLE</td>
<td>343</td>
<td>defines DIMBLK1 as handle to the BLOCK RECORD entry</td>
</tr>
<tr>
<td>DIMBLK2_HANDLE</td>
<td>344</td>
<td>defines DIMBLK2 as handle to the BLOCK RECORD entry</td>
</tr>
<tr>
<td>DIMDEC</td>
<td>271</td>
<td>Sets the number of decimal places displayed for the primary units of a dimension. The precision is based on the units or angle format you have selected.</td>
</tr>
<tr>
<td>DIMDSEP</td>
<td>278</td>
<td>Specifies a single-character decimal separator to use when creating dimensions whose unit format is decimal. When prompted, enter a single character at the Command prompt. If dimension units is set to Decimal, the DIMDSEP character is used instead of the default decimal point. If DIMDSEP is set to NULL (default value, reset by entering a period), the decimal point is used as the dimension separator.</td>
</tr>
<tr>
<td>DIMJUST</td>
<td>280</td>
<td>Controls the horizontal positioning of dimension text.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 = Positions the text above the dimension line and center-justifies it between the extension lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1 = Positions the text next to the first extension line</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 2 = Positions the text next to the second extension line</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 3 = Positions the text above and aligned with the first extension line</td>
</tr>
</tbody>
</table>
Additional DMSYLE Variables DXF R2000

Source: CADDManager Blog
<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIMALTRND</td>
<td>148</td>
<td>Rounds off the alternate dimension units.</td>
</tr>
<tr>
<td>DIMATFIT</td>
<td>289</td>
<td>Determines how dimension text and arrows are arranged when space is not sufficient to place both within the extension lines.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 = Places both text and arrows outside extension lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1 = Moves arrows first, then text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 2 = Moves text first, then arrows</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 3 = Moves either text or arrows, whichever fits best</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A leader is added to moved dimension text when DIMTMOVE is set to 1.</td>
</tr>
<tr>
<td>DIMAZIN</td>
<td>79</td>
<td>Suppresses zeros for angular dimensions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 = Displays all leading and trailing zeros</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1 = Suppresses leading zeros in decimal dimensions (for example, 0.5000 becomes .5000)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 2 = Suppresses trailing zeros in decimal dimensions (for example, 12.5000 becomes 12.5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 3 = Suppresses leading and trailing zeros (for example, 0.5000 becomes .5)</td>
</tr>
<tr>
<td>DIMFRAC</td>
<td>276</td>
<td>Sets the fraction format when DIMLUNIT is set to 4 (Architectural) or 5 (Fractional).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 = Horizontal stacking</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1 = Diagonal stacking</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 2 = Not stacked (for example, 1/2)</td>
</tr>
<tr>
<td>DIMLDRBLK_HANDLE</td>
<td>341</td>
<td>Specifies the arrow type for leaders. Handle to BLOCK RECORD</td>
</tr>
<tr>
<td>DIMLUNIT</td>
<td>277</td>
<td>Sets units for all dimension types except Angular.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1 = Scientific</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 2 = Decimal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 3 = Engineering</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 4 = Architectural (always displayed stacked)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 5 = Fractional (always displayed stacked)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 6 = Microsoft Windows Desktop (decimal format using Control Panel settings for decimal separator and number grouping symbols)</td>
</tr>
</tbody>
</table>

480 Chapter 6. Contents
Text Location

This image shows the default text locations created by BricsCAD for dimension variables \texttt{dimtad} and \texttt{dimjust}:

Unofficial DIMSTYLE Variables for DXF R2007 and later

The following DIMVARS are \textbf{not documented} in the DXF Reference by Autodesk.

<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIMTFILL</td>
<td>69</td>
<td>Text fill 0=off; 1=background color; 2=custom color (see DIMTFILLCLR)</td>
</tr>
<tr>
<td>DIMTFILLCLR</td>
<td>70</td>
<td>Text fill custom color as color index</td>
</tr>
<tr>
<td>DIMFXLON</td>
<td>290</td>
<td>Extension line has fixed length if set to 1</td>
</tr>
<tr>
<td>DIMFXL</td>
<td>49</td>
<td>Length of extension line below dimension line if fixed (DIMFXLON is 1), DIMEXE defines the length above the dimension line</td>
</tr>
<tr>
<td>DIMJOGANG</td>
<td>50</td>
<td>Angle of oblique dimension line segment in jogged radius dimension</td>
</tr>
<tr>
<td>DIML_TYPE_HANDLE</td>
<td>345</td>
<td>Specifies the LINETYPE of the dimension line. Handle to LTYPE table entry</td>
</tr>
<tr>
<td>DIML_TEX1_HANDLE</td>
<td>346</td>
<td>Specifies the LINETYPE of the extension line 1. Handle to LTYPE table entry</td>
</tr>
<tr>
<td>DIML_TEX2_HANDLE</td>
<td>347</td>
<td>Specifies the LINETYPE of the extension line 2. Handle to LTYPE table entry</td>
</tr>
</tbody>
</table>

Extended Settings as Special XDATA Groups

Prior to DXF R2007, some extended settings for the dimension and the extension lines are stored in the XDATA section by following entries, this is not documented by Autodesk:

```
1001
ACAD_DSTYLE_DIM_LINETYPE    <<< linetype for dimension line
1070
380
1005
FFFF
1001
ACAD_DSTYLE_DIM_EXT1_LINETYPE <<< linetype for extension line 1
1070
381
1005
FFFF
1001
ACAD_DSTYLE_DIM_EXT2_LINETYPE <<< linetype for extension line 1
1070
382
1005
FFFF
1001
ACAD_DSTYLE_DIMEXT_ENABLED <<< extension line fixed
1070
383
1070
```

(continues on next page)
1 <<< fixed if 1 else 0
1001
ACAD_DSTYLE_DIMEXT_LENGTH <<< extension line fixed length
1070
378 <<< group code, which differs from R2007 DIMEXLEN
1040
1.33 <<< length of extension line below dimension line

This XDATA groups requires also an appropriate APPID entry in the APPID table. This feature is not supported by ezdxf.

**BLOCK_RECORD Table**

Block records are essential elements for the entities management, each layout (modelspace and paperspace) and every block definition has a block record entry. This block record is the hard owner of the entities of layouts, each entity has an owner handle which points to a block record of the layout.

**DXF Entities**

**DIMENSION Internals**

See also:
- DXF Reference: DIMENSION
- DXFInternals: DIMSTYLE Table

**MLEADER Internals**

See also:
- DXF Reference: MLEADER

Example BricsCAD MultiLeaderContext:

**MTEXT Internals**

The MTEXT entity stores multiline text in a single entity and was introduced in DXF version R13/R14. For more information about the top level stuff go to the MText class.

See also:
- DXF Reference: MTEXT
- ezdxf.entities.MText class

**Orientation**

The MTEXT entity does not establish an OCS. The entity has a text_direction attribute, which defines the local x-axis, the extrusion attribute defines the normal vector and the y-axis = extrusion cross x-axis.
The MTEXT entity can have also a rotation attribute (in degrees), the x-axis attribute has higher priority than the rotation attribute, but it is not clear how to convert the rotation attribute into a text_direction vector, but for most common cases, where only the rotation attribute is present, the extrusion is most likely the WCS z-axis and the rotation is the direction in the xy-plane.

Text Content

The content text is divided across multiple tags of group code 3 and 1, the last line has the group code 1, each line can have a maximum line length of 255 bytes, but BricsCAD (and AutoCAD?) store only 249 bytes in single line and one byte is not always one char.

The text formatting is done by inline codes, see MText class.

Height Calculation

There is no reliable way to calculate the MTEXT height from the existing DXF attributes. The rect_height (group code 43) attribute is not required and seldom present. DXF R2007 introduced the defined_height attribute to store the defined column height of the MTEXT entity but only in column mode. MTEXT entities without columns, except MTEXT entities created with column type “No Columns”, store always 0.0 as defined column height. Which seems to mean: defined by the rendered text content.

The only way to calculate the MTEXT height is to replicate the rendering results of AutoCAD/BricsCAD by implementing a rendering engine for MTEXT.

Width Calculation

The situation for width calculation is better than for the height calculation, but the attributes width and rect_width are not mandatory.

There is a difference between MTEXT entities with and without columns:

Without columns the attribute width (reference column width) contains the true entity width if present. A long word can overshoot this width! The rect_width attribute is seldom present.

For MTEXT with columns, the width attribute is maybe wrong, the correct width for a column is stored in the column_width attribute and the total_width attribute stores the total width of the MTEXT entity overall columns, see also following section “Column Support”.

Background Filling

The background fill support is available for DXF R2007+. The group code 90 defines the kind of background fill:

```
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>off</td>
</tr>
<tr>
<td>1</td>
<td>color defined by group code 63, 421 or 431</td>
</tr>
<tr>
<td>2</td>
<td>drawing window color</td>
</tr>
<tr>
<td>3</td>
<td>background (canvas) color</td>
</tr>
<tr>
<td>16</td>
<td>bit-flag text frame, see Open Design Alliance Specification 20.4.46</td>
</tr>
</tbody>
</table>
```

Group codes to define background fill attributes:
<table>
<thead>
<tr>
<th>45</th>
<th>scaling factor for the border around the text, the value should be in the range of ([1, 5]), where 1 fits exact the MText entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>set the background color by <em>ACI</em>.</td>
</tr>
<tr>
<td>421</td>
<td>set the background color as <em>true color</em> value.</td>
</tr>
<tr>
<td>431</td>
<td>set the background color by color name - no idea how this works</td>
</tr>
<tr>
<td>441</td>
<td>set the transparency of the background fill, not supported by AutoCAD or BricsCAD.</td>
</tr>
</tbody>
</table>

Group codes 45, 90 and 63 are required together if one of them is used. The group code 421 and 431 also requires the group code 63, even this value is ignored.

```plaintext
... <snip>
1 <str> eu feugiat nulla facilisis at vero eros et accumsan et iusto ...
73 <int> 1
44 <float> 1.0
90 <int> 1, b00000001 <<< use a color
63 <int> 1 <<< ACI color (red)
45 <float> 1.5 <<< bg scaling factor, relative to the char height
441 <int> 0 <<< ignored (optional)
... <snip>
```
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet,
Column Support

CAD applications build multiple columns by linking 2 or more MTEXT entities together. In this case each column is a self-sufficient entity in DXF version R13 until R2013. The additional columns specifications are stored in the XDATA if the MTEXT which represents the first column.

DXF R2018 changed the implementation into a single MTEXT entity which contains all the content text at once and stores the column specification in an embedded object.

**Hint:** The width attribute for the linked MTEXT entities could be wrong. Always use the column_width and the total_width attributes in column mode.

There are two column types, the static type has the same column height for all columns, the dynamic type can have the same (auto) height or an individual height for each column.

Common facts about columns for all column types:

- all columns have the same column width
- all columns have the same gutter width
- the top of the column are at the same height

Column Type

The column type defines how a CAD application should create the columns, this is not important for the file format, because the result of this calculation, the column count and the column height, is stored the DXF file.

<table>
<thead>
<tr>
<th>Column Type in BricsCAD</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>All columns have the same height. The “auto height” flag is 0.</td>
</tr>
<tr>
<td>Dynamic (auto height)</td>
<td>Same as the static type, all columns have the same height. The “auto height” flag is 1. The difference to the static type is only important for interactive CAD applications.</td>
</tr>
<tr>
<td>Dynamic (manual height)</td>
<td>same as the dynamic (auto height) type, but each column can have an individual height.</td>
</tr>
<tr>
<td>No column</td>
<td>A regular MTEXT with “defined column height” attribute?</td>
</tr>
</tbody>
</table>

Static Columns R2000

Example for a static column specification:

- Column Type: Static
- Number of Columns: 3
- Height: 150.0, manual entered value and all columns have the same height
- Width: 50.0
- Gutter Width: 12.5
The column height is stored as the “defined column height” in XDATA (46) or the embedded object (41).

DXF R2000 example with a static column specification stored in XDATA:
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam ...
The linked column MTEXT #1B4 in a compressed representation:
The linked MTEXT has no column specification except the “defined column height” in the XDATA. The reference column width is not the real value of 50.0, see XDATA group code 48 in the main MTEXT #9D, instead the total width of 175.0 is stored at group code 41. This is problem if a renderer try to render this MTEXT as a standalone entity. The renderer has to fit the content into the column width by itself and without the correct column width, this will produce an incorrect result.

There exist no back link to the main MTEXT #9D. The linked MTEXT entities appear after the main MTEXT in the layout space, but there can be other entities located between these linked MTEXT entities.

The linked column MTEXT #1B5:

<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;ctrl&gt; MTEXT</td>
</tr>
<tr>
<td>5</td>
<td>&lt;&lt;&lt; entity handle</td>
</tr>
<tr>
<td>9D</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>{ACAD_XDICTIONARY</td>
</tr>
<tr>
<td>360</td>
<td>}</td>
</tr>
<tr>
<td>9F</td>
<td></td>
</tr>
</tbody>
</table>

**Static Columns R2018**

The MTEXT entity in DXF R2018 contains all column information in a single entity. The text content of all three columns are stored in a continuous text string, the separation into columns has to be done by the renderer. The manual column break \N is not used to indicate automatic column breaks. The MTEXT renderer has to replicate the AutoCAD/BricsCAD rendering as exact as possible to achieve the same results, which is very hard without rendering guidelines or specifications.

The example from above in DXF R2018 with a static column specification stored in an embedded object:
<< block record handle of owner layout
1F
100
AcDbEntity
8   << layer
0
100
AcDbMText
10   << (10, 20, 30) insert location in WCS
285.917876152751
20
276.101821192053
30
0.0
40   << character height in drawing units
2.5
41   << reference column width, if not in column mode
62.694536423841
46   << defined column height
150.0
71   << attachment point
1
72   << text flow direction
1
3   << text content of all three columns
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam n...
3
imata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit...
3
a rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lor...
3
vero eos et accusam et justo duo dolores et ea rebum. Stet clita ka...
3
eu feugiat nulla facilisis at vero eros et accumsan et iusto odio s...
3
od tincidunt ut laoreet dolore magna aliquam erat volutpat. \P\PU...
3
e velit esse molestie consequat, vel illum dolore eu feugiat nulla ...
3
obis eleifend option congue nihil imperdiet doming id quod mazim pl...
3
m ad minim veniam, quis nostrud exerci tation ullamcorper suscipit ...
3
lisit. \P\PAt vero eos et accusam et justo duo dolores et ea rebu...
3
labore et dolore magna aliquyam erat, sed diam voluptua. At vero ...
3
litr, At accusam aliquyam diam diam dolore dolores duo eirmod eos e...
1
ipsum dolor sit amet, consetetur
73   << line spacing style
1
44   << line spacing factor
1.0
101   << column specification as embedded object
| Embedded Object | 70 <<<< ???
|----------------|---------------
| 10             | (10, 20, 30) text direction vector (local x-axis)
| 1.0            |
| 20             |
| 0.0            |
| 30             |
| 0.0            |
| 11             | (11, 21, 31) repeated insert location of AcDbMText
| 285.917876152751 |
| 21             |
| 276.101821192053 |
| 31             |
| 0.0            |
| 40             | repeated reference column width
| 62.694536423841 |
| 41             | repeated defined column height
| 150.0          |
| 42             | extents (total) width
| 175.0          |
| 43             | extents (total) height, max. height if different column heights
| 150.0          |
| 71             | column type: 0=no column; 1=static columns; 2=dynamic columns
| 1              |
| 72             | column height count
| 3              |
| 44             | column width
| 50.0           |
| 45             | column gutter width
| 12.5           |
| 73             | flag column auto height
| 0              |
| 74             | flag reversed column flow
| 0              |
| 1001           | AcadAnnotative
| 1000           |
| 1002           |
| {              |
| 1070           |
| 0              |
| 1002           |
| }              |

**Dynamic (auto height) Columns R2000**

Example for a dynamic column specification:

- Column Type: Dynamic
- Number of Columns: 3
- Height: 158.189... adjusted by widget and all columns have the same height
• Width: 50.0
• Gutter Width: 12.5

0 <ctrl> MTEXT
5 <hex> #A2 <<< entity handle
... <snip>
330 <hex> #1F <<< block record handle of owner layout
100 <ctrl> AcDbEntity
8 <str> 0 <<< layer
100 <ctrl> AcDbMText
10 <point> (-133.714579865783, 276.101821192053, 0.0) <<< insert location in WCS
40 <float> 2.5 <<< character height in drawing units
41 <float> 62.694536423841 <<< reference column width, if not in column mode
71 <int> 1 <<< attachment point
72 <int> 1 <<< flag text flow direction
3 <str> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed dia... ... <snip>
73 <int> 1 <<< line spacing style
44 <float> 1.0 <<< line spacing factor
1001 <ctrl> AcadAnnotative
... <snip>
1001 <ctrl> ACAD
1000 <str> ACAD_MTEXT_COLUMN_INFO_BEGIN
1070 <int> 75 <<< column type: 2=dynamic columns
1070 <int> 2
1070 <int> 79 <<< flag column auto height
1070 <int> 1
1070 <int> 76 <<< column count
1070 <int> 3
1070 <int> 78 <<< flag column flow reversed
1070 <int> 0
1070 <int> 48 <<< column width in column mode
1040 <float> 50.0
1070 <int> 49 <<< column gutter width
1040 <float> 12.5
1000 <str> ACAD_MTEXT_COLUMN_INFO_END
1000 <str> ACAD_MTEXT_COLUMNS_BEGIN
1070 <int> 47 <<< column count
1070 <int> 3
1005 <hex> #1B6 <<< handle to 2. column as MTEXT entity
1005 <hex> #1B7 <<< handle to 3. column as MTEXT entity
1000 <str> ACAD_MTEXT_COLUMNS_END
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_BEGIN
1070 <int> 46 <<< defined column height
1040 <float> 158.189308131867
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_END

The linked column MTEXT #1B6:

0 <ctrl> MTEXT
... <snip>
100 <ctrl> AcDbMText
10 <point> (-71.214579865783, 276.101821192053, 0.0)
40 <float> 2.5
41 <float> 175.0 <<< invalid column width
... <snip>
1001 <ctrl> ACAD

(continues on next page)
The linked column MTEXT #1B7:

```
0 <ctrl> MTEXT
  ... <snip>
100 <ctrl> AcDbMText
10 <point> (-8.714579865783, 276.101821192053, 0.0)  <<< insert location in WCS
40 <float> 2.5
41 <float> 175.0  <<< invalid column width
  ... <snip>
1001 <ctrl> ACAD
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_BEGIN
1070 <int> 46  <<< defined column height
1040 <float> 158.189308131867
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_END
```

Dynamic (auto height) Columns R2018

```
0 <ctrl> MTEXT
  ... <snip>
5 <hex> #A2  <<< entity handle
102 <ctrl> {ACAD_XDICTIONARY
360 <hex> #A3
102 <ctrl> }
330 <hex> #1F  <<< block record handle of owner layout
100 <ctrl> AcDbEntity
8 <str> 0  <<< layer
100 <ctrl> AcDbMText
10 <point> (-133.714579865783, 276.101821192053, 0.0)  <<< insert location in WCS
40 <float> 2.5  <<< character height in drawing units
41 <float> 62.694536423841  <<< reference column width, if not in column mode
46 <float> 158.189308131867  <<< defined column height
71 <int> 1  <<< attachment point
72 <int> 1  <<< text flow direction
3 <str> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam...
  ... <snip> text content of all three columns
73 <int> 1  <<< line spacing style
44 <float> 1.0  <<< line spacing factor
101 <ctrl> Embedded Object
70 <int> 1, b00000001  <<< ???
10 <point> (1.0, 0.0, 0.0)  <<< text direction vector (local x-axis)
11 <point> (-133.714579865783, 276.101821192053, 0.0)  <<< repeated insert location
40 <float> 62.694536423841  <<< repeated reference column width
41 <float> 158.189308131867  <<< repeated defined column height
42 <float> 175.0  <<< extents (total) width
43 <float> 158.189308131867  <<< extents (total) height, max. height if different, as column heights
71 <int> 2  <<< column type: 2=dynamic columns
72 <int> 0  <<< column height count
44 <float> 50.0  <<< column width
45 <float> 12.5  <<< column gutter width
```
Dynamic (manual height) Columns R2000

Example for a **dynamic** column specification with manual height definition for three columns with different column heights. None of the (linked) MTEXT entities does contain XDATA for the defined column height.

**Hint:** If “content type” is 2 and flag “column auto height” is 0, no defined height in XDATA.

- Column Type: Dynamic
- Number of Columns: 3
- Height: 164.802450331126, max. column height
- Width: 50.0
- Gutter Width: 12.5
The linked column MTEXT #1B2:

```plaintext
0 <ctrl> MTEXT
... <snip>
100 <ctrl> AcDbMText
10 <point> (132.306121185863, 276.101821192053, 0.0) <<< insert location in WCS
40 <float> 2.5 <<< character height in drawing units
41 <float> 175.0 <<< invalid reference column width
... <snip>
73 <int> 1 <<< line spacing style
44 <float> 1.0 <<< line spacing factor
```

The linked column MTEXT #1B3:

```plaintext
0 <ctrl> MTEXT
... <snip>
100 <ctrl> AcDbMText
10 <point> (132.306121185863, 276.101821192053, 0.0)
40 <float> 2.5
41 <float> 175.0 <<< invalid reference column width
... <snip>
73 <int> 1
44 <float> 1.0
(continues on next page)
... <snip>
100 <ctrl> AcDbMText
10 <point> (194.806121185863, 276.101821192053, 0.0)
40 <float> 2.5
41 <float> 175.0
... <snip>
73 <int> 1
44 <float> 1.0

**Dynamic (manual height) Columns R2018**

**Hint:** If “content type” is 2 and flag “column auto height” is 0, the “defined column height” is 0.0.

```plaintext
0 <ctrl> MTEXT
5 <hex> #9C <<< entity handle
330 <hex> #1F
100 <ctrl> AcDbEntity
8 <str> 0 <<< block record handle of owner layout
100 <ctrl> AcDbMText
10 <point> (69.806121185863, 276.101821192053, 0.0) <<< insert location in WCS
40 <float> 2.5 <<< character height in drawing units
41 <float> 62.694536423841 <<< reference column width, if not in column mode
46 <float> 0.0 <<< defined column height
71 <int> 1 <<< attachment point
72 <int> 1 <<< text flow direction
3 <str> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam...
... <snip> text content of all three columns
73 <int> 1 <<< line spacing style
44 <float> 1.0 <<< line spacing factor
101 <ctrl> Embedded Object
70 <int> 1, b00000001 <<< ???
10 <point> (1.0, 0.0, 0.0) <<< text direction vector (local x-axis)
11 <point> (69.806121185863, 276.101821192053, 0.0) <<< repeated insert location
40 <float> 62.694536423841 <<< repeated reference column width
41 <float> 0.0 <<< repeated defined column height
42 <float> 175.0 <<< extents (total) width
43 <float> 164.802450331126 <<< extents (total) height, max. height if different, column heights
71 <int> 2 <<< column type: 2=dynamic columns
72 <int> 3 <<< column height count
44 <float> 50.0 <<< column width
45 <float> 12.5 <<< column gutter width
73 <int> 0 <<< flag column auto height
74 <int> 0 <<< flag reversed column flow
46 <float> 164.802450331126 <<< column height 1. column
46 <float> 154.311699779249 <<< column height 2. column
46 <float> 0.0 <<< column height 3. column, takes the rest?
```

**No Columns R2000**

I have no idea why this column type exist, but at least provides a reliable value for the MTEXT height by the “defined column height” attribute. The column type is not stored in the MTEXT entity and is therefore not detectable!
• Column Type: No columns
• Number of Columns: 1
• Height: 158.189308131867, defined column height
• Width: 175.0, reference column width

No Columns R2018

Does not contain an embedded object.

DXF Objects

TODO

6.11.3 Management Structures
Block Management Structures

A BLOCK is a layout like the modelspace or a paperspace layout, with the similarity that all these layouts are containers for graphical DXF entities. This block definition can be referenced in other layouts by the INSERT entity. By using block references, the same set of graphical entities can be located multiple times at different layouts, this block references can be stretched and rotated without modifying the original entities. A block is referenced only by its name defined by the DXF tag (2, name), there is a second DXF tag (3, name2) for the block name, which is not further documented by Autodesk, just ignore it.

The (10, base_point) tag in BLOCK defines a insertion point of the block, by ‘inserting’ a block by the INSERT entity, this point of the block is placed at the location defined by the (10, insert) tag in the INSERT entity, and it is also the base point for stretching and rotation.

A block definition can contain INSERT entities, and it is possible to create cyclic block definitions (a BLOCK contains a INSERT of itself), but this should be avoided, CAD applications will not load the DXF file at all or maybe just crash. This is also the case for all other kinds of cyclic definitions like: BLOCK “A” -> INSERT BLOCK “B” and BLOCK “B” -> INSERT BLOCK “A”.

See also:

- ezdxf DXF Internals: BLOCKS Section
- DXF Reference: BLOCKS Section
- DXF Reference: BLOCK Entity
- DXF Reference: ENDBLK Entity
- DXF Reference: INSERT Entity

Block Names

Block names has to be unique and they are case insensitive (“Test” == “TEST”). If there are two or more block definitions with the same name, AutoCAD merges these blocks into a single block with unpredictable properties of all these blocks. In my test with two blocks, the final block has the name of the first block and the base-point of the second block, and contains all entities of both blocks.

Block Definitions in DXF R12

In DXF R12 the definition of a block is located in the BLOCKS section, no additional structures are needed. The definition starts with a BLOCK entity and ends with a ENDBLK entity. All entities between this two entities are the content of the block, the block is the owner of this entities like any layout.

As shown in the DXF file below (created by AutoCAD LT 2018), the BLOCK entity has no handle, but ezdxf writes also handles for the BLOCK entity and AutoCAD doesn’t complain.

DXF R12 BLOCKS structure:

```
0 <<< start of a SECTION
SECTION
2 <<< start of BLOCKS section
BLOCKS
... <<< modelspace and paperspace block definitions not shown,
... <<< see layout management
... 0 <<< start of a BLOCK definition
```

(continues on next page)
8     <<< layer
0
2     <<< block name
ArchTick
70     <<< flags
1
10     <<< base point, x
0.0
20     <<< base point, y
0.0
30     <<< base point, z
0.0
3     <<< second BLOCK name, same as (2, name)
ArchTick
1     <<< xref name, if block is an external reference
      <<< empty string!
0     <<< start of the first entity of the BLOCK
LINE
5
28E
8
0
62
0
10
500.0
20
500.0
30
0.0
11
500.0
21
511.0
31
0.0
0     <<< start of the second entity of the BLOCK
LINE...
0.0
0     <<< ENDBLK entity, marks the end of the BLOCK definition
ENDBLK
5     <<< ENDBLK gets a handle by AutoCAD, but BLOCK didn't
2F2
8
0
0     <<< start of next BLOCK entity
BLOCK...
0     <<< end BLOCK entity
ENDBLK
0     <<< end of BLOCKS section
ENDSEC
Block Definitions in DXF R2000+

The overall organization in the BLOCKS sections remains the same, but additional tags in the BLOCK entity, have to be maintained.

Especially the concept of ownership is important. Since DXF R13 every graphic entity is associated to a specific layout and a BLOCK definition is also a layout. So all entities in the BLOCK definition, including the BLOCK and the ENDBLK entities, have an owner tag (330, ...), which points to a BLOCK_RECORD entry in the BLOCK_RECORD table. This BLOCK_RECORD is the main management structure for all layouts and is the real owner of the layout entities.

As you can see in the chapter about Layout Management Structures, this concept is also valid for modelspace and paperspace layouts, because these layouts are also BLOCKS, with the special difference, that the entities of the modelspace and the active paperspace layout are stored in the ENTITIES section.

See also:

- DXF R13 and later tag structure
- ezdxf DXF Internals: TABLES Section
- DXF Reference: TABLES Section
- DXF Reference: BLOCK_RECORD Entity

DXF R13 BLOCKS structure:

```
0    <<< start of a SECTION
SECTION
2    <<< start of BLOCKS section
BLOCKS
...  <<< modelspace and paperspace block definitions not shown,
...  <<< see layout management
0    <<< start of BLOCK definition
BLOCK
5    <<< even BLOCK gets a handle now ;)
23A
330  <<< owner tag, the owner of a BLOCK is a BLOCK_RECORD in the
...  BLOCK_RECORD table
238
100  <<< subclass marker
AcDbEntity
8    <<< layer of the BLOCK definition
0
100  <<< subclass marker
```
AcDbBlockBegin
2 <<< BLOCK name
ArchTick
70 <<< flags
0
10 <<< base point, x
0.0
20 <<< base point, y
0.0
30 <<< base point, z
0.0
3 <<< second BLOCK name, same as (2, name)
ArchTick
1 <<< xref name, if block is an external reference
    <<< empty string!
0 <<< start of the first entity of the BLOCK
LWPOLYLINE
5
239
330 <<< owner tag of LWPOLYLINE
238 <<< handle of the BLOCK_RECORD!
100
AcDbEntity
8
0
6
ByBlock
62
0
100
AcDbPolyline
90
2
70
0
43
0.15
10
-0.5
20
-0.5
10
0.5
20
0.5
0 <<< ENDBLK entity, marks the end of the BLOCK definition
ENDBLK
5 <<< handle
23B
330 <<< owner tag, same BLOCK_RECORD as for the BLOCK entity
238
100 <<< subclass marker
AcDbEntity
8 <<< ENDBLK requires the same layer as the BLOCK entity!
0
100 <<< subclass marker
AcDbBlockEnd

(continues on next page)
DXF R13 BLOCK_RECORD structure:

```
0  <<< start of a SECTION
SECTION
2  <<< start of TABLES section
TABLES
0  <<< start of a TABLE
TABLE
2  <<< start of the BLOCK_RECORD table
BLOCK_RECORD
5  <<< handle of the table
  1
330 <<< owner tag of the table
  0 <<< is always #0
100 <<< subclass marker
AcDbSymbolTable
70  <<< count of table entries, not reliable
  4
0  <<< start of first BLOCK_RECORD entry
BLOCK_RECORD
5  <<< handle of BLOCK_RECORD, in ezdxf often referred to as "layout key"
  1F
330 <<< owner of the BLOCK_RECORD is the BLOCK_RECORD table
  1
100 <<< subclass marker
AcDbSymbolTableRecord
100 <<< subclass marker
AcDbBlockTableRecord
2 <<< name of the BLOCK or LAYOUT
  *Model_Space
340 <<< pointer to the associated LAYOUT object
4AF
70  <<< AC1021 (R2007) block insertion units
  0
280 <<< AC1021 (R2007) block explodability
  1
281 <<< AC1021 (R2007) block scalability
  0
...
  <<< paperspace not shown
...
0  <<< next BLOCK_RECORD
BLOCK_RECORD
5  <<< handle of BLOCK_RECORD, in ezdxf often referred to as "layout key"
  238
330 <<< owner of the BLOCK_RECORD is the BLOCK_RECORD table
  1
100 <<< subclass marker
```
Layout Management Structures

Layouts are separated entity spaces, there are three different Layout types:

1. modelspace contains the ‘real’ world representation of the drawing subjects in real world units.
2. paperspace layouts are used to create different drawing sheets of the modelspace subjects for printing or PDF export
3. Blocks are reusable sets of graphical entities, inserted/referenced by the INSERT entity.

All layouts have at least a BLOCK definition in the BLOCKS section and since DXF R13 exist the BLOCK_RECORD table with an entry for every BLOCK in the BLOCKS section.

See also:
Information about Block Management Structures

The name of the modelspace BLOCK is “*Model_Space” (DXF R12: “$MODEL_SPACE”) and the name of the active paperspace BLOCK is “*Paper_Space” (DXF R12: “SPAPER_SPACE”), the entities of these two layouts are stored in the ENTITIES section, DXF R12 supports just one paperspace layout.

DXF R13+ supports multiple paperspace layouts, the active layout is still called “*Paper_Space”, the additional inactive paperspace layouts are named by the scheme “*Paper_Spacennnn”, where the first inactive paper space is called “*Paper_Space0”, the second “*Paper_Space1” and so on. A none consecutive numbering is tolerated by AutoCAD. The content of the inactive paperspace layouts are stored as BLOCK content in the BLOCKS section. These names are just the DXF internal layout names, each layout has an additional layout name which is displayed to the user by the CAD application.

A BLOCK definition and a BLOCK_RECORD is not enough for a proper layout setup, an LAYOUT entity in the OBJECTS section is also required. All LAYOUT entities are managed by a DICTIONARY entity, which is referenced as “ACAD_LAYOUT” entity in the root DICTIONARY of the DXF file.
Note: All floating point values are rounded to 2 decimal places for better readability.

### LAYOUT Entity

Since DXF R2000 modelspace and paperspace layouts require the DXF LAYOUT entity.

```
0
LAYOUT
5 <<< handle
59
102 <<< extension dictionary (ignore)
{ACAD_XDICTIONARY
360
1C3
102
}
102 <<< reactor (required?)
{ACAD_REACTORS
330
1A <<< pointer to "ACAD_LAYOUT" DICTIONARY (layout management table)
102
}
330 <<< owner handle
1A <<< pointer to "ACAD_LAYOUT" DICTIONARY (same as reactor pointer)
100 <<< PLOTSETTINGS
AcDbPlotSettings
1 <<< page setup name
2 <<< name of system printer or plot configuration file
none_device
4 <<< paper size, part in braces should follow the schema
... (width_x_height_unit) unit is 'Inches' or 'MM'
... Letter\(8.50_x_11.00\_Inches\) the part in front of the braces is
... ignored by AutoCAD
6 <<< plot view name
40 <<< size of unprintable margin on left side of paper in millimeters,
... defines also the plot origin-x
6.35
41 <<< size of unprintable margin on bottom of paper in millimeters,
... defines also the plot origin-y
6.35
42 <<< size of unprintable margin on right side of paper in millimeters
6.35
43 <<< size of unprintable margin on top of paper in millimeters
6.35
44 <<< plot paper size: physical paper width in millimeters
215.90
45 <<< plot paper size: physical paper height in millimeters
279.40
46 <<< X value of plot origin offset in millimeters, moves the plot origin-x
0.0
47 <<< Y value of plot origin offset in millimeters, moves the plot origin-y
0.0
```

(continues on next page)
ezdxf Documentation, Release 0.16.2

(continued from previous page)

```python
48 <<< plot window area: X value of lower-left window corner
0.0
49 <<< plot window area: Y value of lower-left window corner
0.0
140 <<< plot window area: X value of upper-right window corner
0.0
141 <<< plot window area: Y value of upper-right window corner
0.0
142 <<< numerator of custom print scale: real world (paper) units, 1.0
... for scale 1:50
1.0
143 <<< denominator of custom print scale: drawing units, 50.0
... for scale 1:50
1.0
70 <<< plot layout flags, bit-coded (... too many options)
688 <<< b1010110000 = UseStandardScale(16)/PlotPlotStyle(32)
... PrintLineweights(128)/DrawViewportsFirst(512)
72 <<< plot paper units (0/1/2 for inches/millimeters/pixels), are pixels really supported?
0
73 <<< plot rotation (0/1/2/3 for 0deg/90deg counter-cw/upside-down/90deg cw)
1 <<< 90deg clockwise
74 <<< plot type 0-5 (... too many options)
5 <<< 5 = layout information
7 <<< current plot style name, e.g. 'acad.ctb' or 'acadlt.ctb'
75 <<< standard scale type 0-31 (... too many options)
16 <<< 16 = 1:1, also 16 if user scale type is used
147 <<< unit conversion factor
1.0 <<< for plot paper units in mm, else 0.03937... (1/25.4) for inches ... as plot paper units
76 <<< shade plot mode (0/1/2/3 for as displayed/wireframe/hidden/rendered)
0 <<< as displayed
77 <<< shade plot resolution level 1-5 (... too many options)
2 <<< normal
78 <<< shade plot custom DPI: 100-32767, Only applied when shade plot ... resolution level is set to 5 (Custom)
300
148 <<< paper image origin: X value
0.0
149 <<< paper image origin: Y value
0.0
100 <<< LAYOUT settings
AcDbLayout
1 <<< layout name
Layout1
70 <<< flags bit-coded
1 <<< 1 = Indicates the PSLTSSCALE value for this layout when this ... layout is current
71 <<< Tab order ("Model" tab always appears as the first tab ... regardless of its tab order)
1
10 <<< minimum limits for this layout (defined by LIMMIN while this ... layout is current)
-0.25 <<< x value, distance of the left paper margin from the plot ... origin-x, in plot paper units and by scale (e.g. x50 for 1:50)
20 <<< group code for y value
```

(continues on next page)
-0.25 <<< y value, distance of the bottom paper margin from the plot
... origin-y, in plot paper units and by scale (e.g. x50 for 1:50)
11 <<< maximum limits for this layout (defined by LIMMAX while this
... layout is current)
10.75 <<< x value, distance of the right paper margin from the plot
... origin-x, in plot paper units and by scale (e.g. x50 for 1:50)
21 <<< group code for y value
8.25 <<< y value, distance of the top paper margin from the plot
... origin-y, in plot paper units and by scale (e.g. x50 for 1:50)
12 <<< insertion base point for this layout (defined by INSBASE while
... this layout is current)
0.0 <<< x value
22 <<< group code for y value
0.0 <<< y value
32 <<< group code for z value
0.0 <<< z value
14 <<< minimum extents for this layout (defined by EXTMIN while this
... layout is current), AutoCAD default is (1e20, 1e20, 1e20)
1.05 <<< x value
24 <<< group code for y value
0.80 <<< y value
34 <<< group code for z value
0.0 <<< z value
15 <<< maximum extents for this layout (defined by EXTMAX while this
... layout is current), AutoCAD default is (-1e20, -1e20, -1e20)
9.45 <<< x value
25 <<< group code for y value
7.20 <<< y value
35 <<< group code for z value
0.0 <<< z value
146 <<< elevation ???
0.0
13 <<< UCS origin (3D Point)
0.0 <<< x value
23 <<< group code for y value
0.0 <<< y value
33 <<< group code for z value
0.0 <<< z value
16 <<< UCS X-axis (3D vector)
1.0 <<< x value
26 <<< group code for y value
0.0 <<< y value
36 <<< group code for z value
0.0 <<< z value
17 <<< UCS Y-axis (3D vector)
0.0 <<< x value
27 <<< group code for y value
1.0 <<< y value
37 <<< group code for z value
0.0 <<< z value
76 <<< orthographic type of UCS 0-6 (... too many options)
0 <<< 0 = UCS is not orthographic ???
330 <<< ID/handle of required block table record
58
331 <<< ID/handle to the viewport that was last active in this layout
... when the layout was current
1B9
(continues on next page)
And as it seems this is also not enough for a well defined LAYOUT, at least a “main” VIEWPORT entity with ID=1 is required for paperspace layouts, located in the entity space of the layout.

The modelspace layout requires (?) a VPORT entity in the VPORT table (group code 331 in the AcDbLayout subclass).

**Main VIEWPORT Entity for LAYOUT**

The “main” viewport for layout “Layout1” shown above. This viewport is located in the associated BLOCK definition called “*Paper_Space0*”. Group code 330 in subclass AcDbLayout points to the BLOCK_RECORD of “*Paper_Space0*”.

Remember: the entities of the *active* paperspace layout are located in the ENTITIES section, therefore “Layout1” is not the active paperspace layout.

The “main” VIEWPORT describes, how the application shows the paperspace layout on the screen, and I guess only AutoCAD needs this values.

```
1001 <<< extended data (ignore)
...
```

```
0  VIEWPORT
  5 <<< handle
```
1B4
102  <<< extension dictionary (ignore)
\{ACAD_XDICTI\n360
1B5
102
\}
330  <<< owner handle
58  <<< points to BLOCK_RECORD (same as group code 330 in AcDbLayout of
... "Layout1")
100
AcDbEntity
67  <<< paperspace flag
  1  <<< 0 = modelspace; 1 = paperspace
  8  <<< layer,
  0
100
AcDbViewport
10  <<< Center point (in WCS)
  5.25  <<< x value
  20  <<< group code for y value
  4.00  <<< y value
  30  <<< group code for z value
  0.0  <<< z value
  40  <<< width in paperspace units
  23.55  <<< VIEW size in AutoCAD, depends on the workstation configuration
  41  <<< height in paperspace units
  9.00  <<< VIEW size in AutoCAD, depends on the workstation configuration
  68  <<< viewport status field -1/0/n
  2  <<< >0 On and active. The value indicates the order of stacking for
... the viewports, where 1 is the active viewport, 2 is the next, and so forth
  69  <<< viewport ID
  1  <<< "main" viewport has always ID=1
  12  <<< view center point in Drawing Coordinate System (DCS), defines
... the center point of the VIEW in relation to the LAYOUT origin
  5.25  <<< x value
  22  <<< group code for y value
  4.00  <<< y value
  13  <<< snap base point in modelspace
  0.0  <<< x value
  23  <<< group code for y value
  0.0  <<< y value
  14  <<< snap spacing in modelspace units
  0.5  <<< x value
  24  <<< group code for y value
  0.5  <<< y value
  15  <<< grid spacing in modelspace units
  0.5  <<< x value
  25  <<< group code for y value
  0.5  <<< y value
  16  <<< view direction vector from target (in WCS)
  0.0  <<< x value
  26  <<< group code for y value
  0.0  <<< y value
  36  <<< group code for z value
  1.0  <<< z value
  17  <<< view target point
(continues on next page)
(continued from previous page)

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>x value</td>
</tr>
<tr>
<td>27</td>
<td>group code for y value</td>
</tr>
<tr>
<td>0.0</td>
<td>y value</td>
</tr>
<tr>
<td>37</td>
<td>group code for z value</td>
</tr>
<tr>
<td>0.0</td>
<td>z value</td>
</tr>
<tr>
<td>42</td>
<td>perspective lens length, focal length?</td>
</tr>
<tr>
<td>50.0</td>
<td>50mm</td>
</tr>
<tr>
<td>43</td>
<td>front clip plane z value</td>
</tr>
<tr>
<td>0.0</td>
<td>z value</td>
</tr>
<tr>
<td>44</td>
<td>back clip plane z value</td>
</tr>
<tr>
<td>0.0</td>
<td>z value</td>
</tr>
<tr>
<td>45</td>
<td>view height (in modelspace units)</td>
</tr>
<tr>
<td>9.00</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>snap angle</td>
</tr>
<tr>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>view twist angle</td>
</tr>
<tr>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>circle zoom percent</td>
</tr>
<tr>
<td>90</td>
<td>Viewport status bit-coded flags (... too many options)</td>
</tr>
<tr>
<td>819232</td>
<td>b1001000000000100000</td>
</tr>
<tr>
<td>1</td>
<td>plot style sheet name assigned to this viewport</td>
</tr>
<tr>
<td>281</td>
<td>render mode (... too many options)</td>
</tr>
<tr>
<td>0</td>
<td>0 = 2D optimized (classic 2D)</td>
</tr>
<tr>
<td>71</td>
<td>UCS per viewport flag</td>
</tr>
<tr>
<td>1</td>
<td>1 = This viewport stores its own UCS which will become the current UCS whenever the viewport is activated</td>
</tr>
<tr>
<td>74</td>
<td>Display UCS icon at UCS origin flag</td>
</tr>
<tr>
<td>0</td>
<td>this field is currently being ignored and the icon always represents the viewport UCS</td>
</tr>
<tr>
<td>110</td>
<td>UCS origin (3D point)</td>
</tr>
<tr>
<td>0.0</td>
<td>x value</td>
</tr>
<tr>
<td>120</td>
<td>group code for y value</td>
</tr>
<tr>
<td>0.0</td>
<td>y value</td>
</tr>
<tr>
<td>130</td>
<td>group code for z value</td>
</tr>
<tr>
<td>0.0</td>
<td>z value</td>
</tr>
<tr>
<td>111</td>
<td>UCS X-axis (3D vector)</td>
</tr>
<tr>
<td>1.0</td>
<td>x value</td>
</tr>
<tr>
<td>121</td>
<td>group code for y value</td>
</tr>
<tr>
<td>0.0</td>
<td>y value</td>
</tr>
<tr>
<td>131</td>
<td>group code for z value</td>
</tr>
<tr>
<td>0.0</td>
<td>z value</td>
</tr>
<tr>
<td>112</td>
<td>UCS Y-axis (3D vector)</td>
</tr>
<tr>
<td>0.0</td>
<td>x value</td>
</tr>
<tr>
<td>122</td>
<td>group code for y value</td>
</tr>
<tr>
<td>1.0</td>
<td>y value</td>
</tr>
<tr>
<td>132</td>
<td>group code for z value</td>
</tr>
<tr>
<td>0.0</td>
<td>z value</td>
</tr>
<tr>
<td>79</td>
<td>Orthographic type of UCS (... too many options)</td>
</tr>
<tr>
<td>0</td>
<td>0 = UCS is not orthographic</td>
</tr>
<tr>
<td>146</td>
<td>elevation</td>
</tr>
<tr>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>170</td>
<td>shade plot mode (0/1/2/3 for as displayed/wireframe/hidden/rendered)</td>
</tr>
<tr>
<td>0</td>
<td>as displayed</td>
</tr>
<tr>
<td>61</td>
<td>frequency of major grid lines compared to minor grid lines</td>
</tr>
<tr>
<td>5</td>
<td>major grid subdivided by 5</td>
</tr>
</tbody>
</table>
6.12 Developer Guides

Information about ezdxf internals.

6.12.1 Design

The Package Design for Developers section shows the structure of the ezdxf package for developers with more experience, which want to have more insight into the package an maybe want to develop add-ons or want contribute to the ezdxf package.

!!! UNDER CONSTRUCTION !!!

Package Design for Developers

A DXF document is divided into several sections, this sections are managed by the Drawing object. For each section exist a corresponding attribute in the Drawing object:

<table>
<thead>
<tr>
<th>Section</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEADER</td>
<td>Drawing.header</td>
</tr>
<tr>
<td>CLASSES</td>
<td>Drawing.classes</td>
</tr>
<tr>
<td>TABLES</td>
<td>Drawing.tables</td>
</tr>
<tr>
<td>BLOCKS</td>
<td>Drawing.blocks</td>
</tr>
<tr>
<td>ENTITIES</td>
<td>Drawing.entities</td>
</tr>
<tr>
<td>OBJECTS</td>
<td>Drawing.objects</td>
</tr>
</tbody>
</table>

Resource entities (LAYER, STYLE, LTYPE, ...) are stored in tables in the TABLES section. A table owns the table entries, the owner handle of table entry is the handle of the table. Each table has a shortcut in the Drawing object:
### Table

<table>
<thead>
<tr>
<th>Table</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPID</td>
<td>Drawing.appids</td>
</tr>
<tr>
<td>BLOCK_RECORD</td>
<td>Drawing.block_records</td>
</tr>
<tr>
<td>DIMSTYLE</td>
<td>Drawing.dimstyles</td>
</tr>
<tr>
<td>LAYER</td>
<td>Drawing.layers</td>
</tr>
<tr>
<td>LTYPE</td>
<td>Drawing.linetypes</td>
</tr>
<tr>
<td>STYLE</td>
<td>Drawing.styles</td>
</tr>
<tr>
<td>UCS</td>
<td>Drawing.ucs</td>
</tr>
<tr>
<td>VIEW</td>
<td>Drawing.views</td>
</tr>
<tr>
<td>VPORT</td>
<td>Drawing.viewports</td>
</tr>
</tbody>
</table>

Graphical entities are stored in layouts: *Modelspace, Paperspace* layouts and *BlockLayout*. The core management object of this layout is the BLOCK_RECORD entity (*BlockRecord*), the BLOCK_RECORD is the real owner of the entities, the owner handle of the entities is the handle of the BLOCK_RECORD and the BLOCK_RECORD also owns and manages the entity space of the layout which contains all entities of the layout.

For more information about layouts see also: *Layout Management Structures*

For more information about blocks see also: *Block Management Structures*

Non-graphical entities (objects) are stored in the OBJECTS section. Every object has a parent object in the OBJECTS section, most likely a DICTIONARY object, and is stored in the entity space of the OBJECTS section.

For more information about the OBJECTS section see also: *OBJECTS Section*

All table entries, DXF entities and DXF objects are stored in the entities database accessible as `Drawing.entitydb`. The entity database is a simple key, value storage, key is the entity handle, value is the DXF object.

For more information about the DXF data model see also: *Data Model*

### Terminology

#### States

DXF entities and objects can have different states:

- **UNBOUND** Entity is not stored in the `Drawing` entity database and DXF attribute `handle` is `None` and attribute `doc` can be `None`

- **BOUND** Entity is stored in the `Drawing` entity database, attribute `doc` has a reference to `Drawing` and DXF attribute `handle` is not `None`

- **UNLINKED** Entity is not linked to a layout/owner, DXF attribute `owner` is `None`

- **LINKED** Entity is linked to a layout/owner, DXF attribute `owner` is not `None`

- **Virtual Entity** State: UNBOUND & UNLINKED

- **Unlinked Entity** State: BOUND & UNLINKED

- **Bound Entity** State: BOUND & LINKED

#### Actions

- **NEW** Create a new DXF document

- **LOAD** Load a DXF document from an external source
CREATE Create DXF structures from NEW or LOAD data
DESTROY Delete DXF structures
BIND Bind an entity to a Drawing, set entity state to BOUND & UNLINKED and check or create required resources
UNBIND unbind ...
LINK Link an entity to an owner/layout. This makes an entity to a real DXF entity, which will be exported at the saving process. Any DXF entity can only be linked to one parent entity like DICTIONARY or BLOCK_RECORD.
UNLINK unlink ...

Loading a DXF Document

Loading a DXF document from an external source, creates a newDrawing object. This loading process has two stages:

First Loading Stage

- LOAD content from external source as SectionDict: loader.load_dxf_structure()
- LOAD tag structures as DXFEntity objects: loader.load_dxf_entities()
- BIND entities: loader.load_and_bind_dxf_content(); Special handling of the BIND process, because the Drawing is not full initialized, a complete validation is not possible at this stage.

Second Loading Stage

Parse SectionDict:
- CREATE sections: HEADER, CLASSES, TABLES, BLOCKS and OBJECTS
- CREATE layouts: Blocks, Layouts
- LINK entities to a owner/layout

The ENTITIES section is a relict from older DXF versions and has to be exported including the modelspace and active paperspace entities, but all entities reside in a BLOCK definition, even modelspace and paperspace layouts are only BLOCK definitions and ezdxf has no explicit ENTITIES section.

Source Code: as developer start your journey at ezdxf.document.Drawing.read(), which has no public documentation, because package-user should use ezdxf.read() and ezdxf.readfile().

New DXF Document

Creating New DXF Entities

The default constructor of each entity type creates a new virtual entity:
- DXF attribute owner is None
- DXF attribute handle is None
- Attribute doc is None
The DXFEntity.new() constructor creates entities with given owner, handle and doc attributes, if doc is not None and entity is not already bound to a document, the new() constructor automatically bind the entity to the given document doc.

There exist only two scenarios:

1. UNBOUND: doc is None and handle is None
2. BOUND: doc is not None and handle is not None

Factory functions

- new(), create a new virtual DXF object/entity
- load(), load (create) virtual DXF object/entity from DXF tags
- bind(), bind an entity to a document, create required resources if necessary (e.g. ImageDefReactor, SEQEND) and raise exceptions for non-existing resources.
  - Bind entity loaded from an external source to a document, all referenced resources must exist, but try to repair as many flaws as possible because errors were created by another application and are not the responsibility of the package-user.
  - Bind an entity from another DXF document, all invalid resources will be removed silently or created (e.g. SEQEND). This is a simple import from another document without resource import, for a more advanced import including resources exist the importer add-on.
  - Bootstrap problem for binding loaded table entries and objects in the OBJECTS section! Can’t use Auditor to repair this objects, because the DXF document is not fully initialized.
- is_bound() returns True if entity is bound to document doc
- unbind() function to remove an entity from a document and set state to a virtual entity, which should also UNLINK the entity from layout, because an layout can not store a virtual entity.
- cls(), returns the class
- register_entity(), registration decorator
- replace_entity(), registration decorator

Class Interfaces

DXF Entities

- NEW constructor to create an entity from scratch
- LOAD constructor to create an entity loaded from an external source
- DESTROY interface to kill an entity, set entity state to dead, which means entity.is_alive returns False. All entity iterators like EntitySpace, EntityQuery, and EntityDB must filter (ignore) dead entities. Calling DXFEntity.destroy() is a regular way to delete entities.
- LINK an entity to a layout by BlockRecord.link(), which set the owner handle to BLOCK_RECORD handle (= layout key) and add the entity to the entity space of the BLOCK_RECORD and set/clear the paperspace flag.
DXF Objects

- NEW, LOAD, DESTROY see DXF entities
- LINK: Linking an DXF object means adding the entity to a parent object in the OBJECTS section, most likely a DICTIONARY object, and adding the object to the entity space of the OBJECTS section, the root-dict is the only entity in the OBJECTS section which has an invalid owner handle “0”. Any other object with an invalid or destroyed owner is an orphaned entity. The audit process destroys and removes orphaned objects.
- Extension dictionaries (ACAD_XDICTIONARY) are DICTIONARY objects located in the OBJECTS sections and can reference/own other entities of the OBJECTS section.
- The root-dictionary is the only entity in the OBJECTS section which has an invalid owner handle “0”. Any other object with an invalid or destroyed owner is an orphaned entity.

Layouts

- LINK interface to link an entity to a layout
- UNLINK interface to remove an entity from a layout

Database

- BIND interface to add an entity to the database of a document
- delete_entity() interface, same as UNBIND and DESTROY an entity

6.12.2 Internal Modules

Fonts

The module `ezdxf.tools.fonts` manages the internal usage of fonts and has no relation how the DXF formats manages text styles.

See also:
The Textstyle entity, the DXF way to define fonts.

The tools in this module provide abstractions to get font measurements with and without the optional Matplotlib package.

For a proper text rendering the font measurements are required. Ezdxf has a lean approach to package dependencies, therefore the rendering results without support from the optional Matplotlib package are not very good.

Font Classes

```python
ezdxf.tools.fonts.make_font(ttf_path: str, cap_height: float, width_factor: float = 1.0) →
```

Factory function to create a font abstraction.

Creates a MatplotlibFont if the Matplotlib font support is available and enabled or else a MonospaceFont.

Parameters

- **ttf_path** – raw font file name as stored in the Textstyle entity
• **cap_height** – desired cap height in drawing units.

• **width_factor** – horizontal text stretch factor

```python
class ezdxf.tools.fonts.MonospaceFont (cap_height: float, width_factor: float = 1.0, baseline: float = 0, descender_factor: float = 0.333, x_height_factor: float = 0.666)
```

Defines a monospaced font without knowing the real font properties. Each letter has the same cap- and descender height and the same width. This font abstraction is used if no Matplotlib font support is available.

Use the `make_font()` factory function to create a font abstraction.

```python
text_width (text: str) → float

Returns the text width in drawing units for the given text based on a simple monospaced font calculation.
```

```python
class ezdxf.tools.fonts.MatplotlibFont (ttf_path: str, cap_height: float = 1.0, width_factor: float = 1.0)
```

This class provides proper font measurement support by using the optional Matplotlib font support.

Use the `make_font()` factory function to create a font abstraction.

```python
text_width (text: str) → float

Returns the text with in drawing units for the given text string. Text rendering and width calculation is done by the Matplotlib `TextPath` class.
```

**Font Anatomy**

- A Visual Guide to the Anatomy of Typography: https://visme.co/blog/type-anatomy/
- Anatomy of a Character: https://www.fonts.com/content/learning/fontology/level-1/type-anatomy/anatomy

**Font Properties**

The default way of DXF to store fonts in the `Textstyle` entity by using the raw TTF file name is not the way how most render backends select fonts.

The render backends and web technologies select the fonts by their properties. This list shows the Matplotlib properties:

- **family** List of font names in decreasing order of priority. The items may include a generic font family name, either “serif”, “sans-serif”, “cursive”, “fantasy”, or “monospace”.

- **style** “normal” (“regular”), “italic” or “oblique”

- **stretch** A numeric value in the range 0-1000 or one of “ultra-condensed”, “extra-condensed”, “condensed”, “semi-condensed”, “normal”, “semi-expanded”, “expanded”, “extra-expanded” or “ultra-expanded”


This way the backend can choose a similar font if the original font is not available.

**See also:**

- PyQt: https://doc.qt.io/archives/qtforpython-5.12/PySide2/QtGui/QFont.html
- W3C: https://www.w3.org/TR/2018/REC-css-fonts-3-20180920/

```python
class ezdxf.tools.fonts.FontFace (ttf, family, style, stretch, weight)
```

This is the equivalent to the Matplotlib `FontProperties` class.
ttf
Raw font file name as string, e.g. “arial.ttf”

family
Family name as string, the default value is “sans-serif”

style
Font style as string, the default value is “normal”

stretch
Font stretch as string, the default value is “normal”

weight
Font weight as string, the default value is “normal”

class ezdxf.tools.fonts.FontMeasurements
See Font Anatomy for more information.

baseline
cap_height
x_height
descender_height
scale (factor: float = 1.0) → ezdxf.tools.fonts.FontMeasurements
scale_from_baseline (desired_cap_height: float) → ezdxf.tools.fonts.FontMeasurements
shift (distance: float = 0.0) → ezdxf.tools.fonts.FontMeasurements

Font Caching

Ezdxf uses Matplotlib to manage fonts and caches the collected information. The default installation of ezdxf provides a basic set of font properties. It is possible to create your own font cache specific for your system: see ezdxf.options.font_cache_directory

The font cache is loaded automatically at startup, if not disabled by setting environment variable EZDXF_AUTO_LOAD_FONTS to False: see Environment Variables

ezdxf.tools.fonts.get_font_face (ttf_path: str, map_shx=True) → ezdxf.tools.fonts.FontFace
Get cached font face definition by TTF file name e.g. “Arial.ttf”.

This function translates a DXF font definition by the raw TTF font file name into a FontFace object. Fonts which are not available on the current system gets a default font face.

Parameters

• ttf_path – raw font file name as stored in the Textstyle entity
• map_shx – maps SHX font names to TTF replacement fonts, e.g. “TXT” -> “txt_____.ttf”

ezdxf.tools.fonts.get_font_measurements (ttf_path: str, map_shx=True) → ezdxf.tools.fonts.FontMeasurements
Get cached font measurements by TTF file name e.g. “Arial.ttf”.

Parameters

• ttf_path – raw font file name as stored in the Textstyle entity
• map_shx – maps SHX font names to TTF replacement fonts, e.g. “TXT” -> “txt_____.ttf”
ezdxf Documentation, Release 0.16.2

```
from ezdxf import tools, fonts

def build_system_font_cache(*, path=None, rebuild=True) -> None:
    """Build system font cache and save it to directory path if given. Set rebuild to False to just add new fonts. Requires the Matplotlib package!"

    A rebuild has to be done only after a new ezdxf installation, or new fonts were added to your system (which you want to use), or an update of ezdxf if you don’t use your own external font cache directory.

    See also: ezdxf.options.font_cache_directory
```

```
def load(path=None, reload=False):
    """Load all caches from given path or from default location, defined by ezdxf.options.font_cache_directory or the default cache from the ezdxf.tools folder.

    This function is called automatically at startup if not disabled by environment variable EZDXF_AUTO_LOAD_FONTS.
```

```
def save(path=None):
    """Save all caches to given path or to default location, defined by options.font_cache_directory or into the ezdxf.tools folder.
```

6.12.3 Internal Data Structures

Entity Database

The `EntityDB` is a simple key/value database to store `DXFEntity` objects by it’s handle, every `Drawing` has its own `EntityDB`, stored in the `Drawing` attribute `entitydb`.

Every DXF entity/object, except tables and sections, are represented as `DXFEntity` or inherited types, this entities are stored in the `EntityDB`, database-key is the `dxf.handle` as plain hex string.

All iterators like `keys()`, `values()`, `items()` and `__iter__()` do not yield destroyed entities.

```
Warning: The `get()` method and the index operator `[]`, return destroyed entities and entities from the trashcan.
```

```
class ezdxf.entitydb.EntityDB:
    __getitem__(handle: str) -> DXFEntity
        Get entity by `handle`, does not filter destroyed entities nor entities in the trashcan.
    __setitem__(handle: str, entity: DXFEntity) -> None
        Set entity for `handle`.
    __delitem__(handle: str) -> None
        Delete entity by `handle`. Removes entity only from database, does not destroy the entity.
    __contains__(item: Union[str, DXFEntity]) -> bool
        True if database contains `handle`.
    __len__() -> int
        Count of database items.
    __iter__() -> Iterable[str]
        Iterable of all handles, does filter destroyed entities but not entities in the trashcan.
    get(handle: str) -> Optional[DXFEntity]
        Returns entity for `handle` or `None` if no entry exist, does not filter destroyed entities.
```
next_handle() \to \text{str}
Returns next unique handle.

keys() \to \text{Iterable[\text{str}]}
Iterable of all handles, does filter destroyed entities.

values() \to \text{Iterable[\text{DXFEntity}]}
Iterable of all entities, does filter destroyed entities.

items() \to \text{Iterable[\text{Tuple[\text{str}, \text{DXFEntity}]]]}
Iterable of all (handle, entities) pairs, does filter destroyed entities.

add\left(\text{entity: \text{DXFEntity}}\right) \to \text{None}
Add entity to database, assigns a new handle to the entity if entity.dxf.handle is None. Adding the same entity multiple times is possible and creates only a single database entry.

new_trashcan() \to \text{ezdxf.entitydb.EntityDB.Trashcan}
Returns a new trashcan, empty trashcan manually by: :func:Trashcan.clear().

trashcan() \to \text{ezdxf.entitydb.EntityDB.Trashcan}
Returns a new trashcan in context manager mode, trashcan will be emptied when leaving context.

purge() \to \text{None}
Remove all destroyed entities from database, but does not empty the trashcan.

Entity Space

class \text{ezdxf.entitydb.EntitySpace} \left(\text{entities=\text{None}}\right)
An EntitySpace is a collection of \text{DXFEntity objects, that stores only references to DXFEntity objects.}
The Modelspace, any Paperspace layout and BlockLayout objects have an EntitySpace container to store their entities.

__iter__() \to \text{Iterable[\text{DXFEntity}]}
Iterable of all entities, filters destroyed entities.

__getitem__(\text{index}) \to \text{DXFEntity}
Get entity at index item

EntitySpace has a standard Python list like interface, therefore index can be any valid list indexing or slicing term, like a single index layout[-1] to get the last entity, or an index slice layout[:10] to get the first 10 or less entities as List[DXFEntity]. Does not filter destroyed entities.

__len__() \to \text{int}
Count of entities including destroyed entities.

has_handle\left(\text{handle: \text{str}}\right) \to \text{bool}
True if handle is present, does filter destroyed entities.

purge()
Remove all destroyed entities from entity space.

add\left(\text{entity: \text{DXFEntity}}\right) \to \text{None}
Add entity.

extend\left(\text{entities: \text{Iterable[\text{DXFEntity}]}}\right) \to \text{None}
Add multiple entities.

remove\left(\text{entity: \text{DXFEntity}}\right) \to \text{None}
Remove entity.
clear() → None
Remove all entities.

**DXF Types**

Required DXF tag interface:
- property `code`: group code as int
- property `value`: tag value of unspecific type
- `dxfstr()`: returns the DXF string
- `clone()`: returns a deep copy of tag

**DXFTag Factory Functions**

`ezdxf.lldxf.types.dxftag(code: int, value: TagValue) → ezdxf.lldxf.types.DXFTag`
DXF tag factory function.

**Parameters**
- `code` – group code
- `value` – tag value

**Returns**: `DXFTag` or inherited

`ezdxf.lldxf.types.tuples_to_tags(iterable: Iterable[Tuple[int, TagValue]]) → Iterable[ezdxf.lldxf.types.DXFTag]`
Returns an iterable if :class: `DXFTag` or inherited, accepts an iterable of (code, value) tuples as input.

**DXFTag**

`class ezdxf.lldxf.types.DXFTag(code: int, value: TagValue)`
Immutable DXFTag class - immutable by design, not by implementation.

**Parameters**
- `code` – group code as int
- `value` – tag value, type depends on group code

**Variables**
- `code` – group code as int (do not change)
- `value` – tag value (read-only property)

**__eq__(other) → bool**
True if `other` and `self` has same content for `code` and `value`.

**__getitem__(index: int)**
Returns `code` for index 0 and `value` for index 1, emulates a tuple.

**__hash__()**
Hash support, `DXFTag` can be used in sets and as dict key.

**__iter__() → Iterable[T_co]**
Returns (code, value) tuples.
__repr__() → str
Returns representation string 'DXFTag(code, value)'.

__str__() → str
Returns content string '(code, value)'.

clon() → ezdxf.llldxf.types.DXFTag
Returns a clone of itself, this method is necessary for the more complex (and not immutable) DXF tag types.

dxfstr() → str
Returns the DXF string e.g. ' 0
LINE
'

DXFBinaryTag
class ezdxf.llldxf.types.DXFBinaryTag(DXFTag)
 Immutable BinaryTags class - immutable by design, not by implementation.

dxfstr() → str
Returns the DXF string for all vertex components.

tostring() → str
Returns binary value as single hex-string.

DXFVertex
class ezdxf.llldxf.types.DXFVertex(DXFTag)
 Represents a 2D or 3D vertex, stores only the group code of the x-component of the vertex, because the y-group-code is x-group-code + 10 and z-group-code id x-group-code+20, this is a rule that ALWAYS applies. This tag is immutable by design, not by implementation.

Parameters

• code – group code of x-component
• value – sequence of x, y and optional z values

dxfstr() → str
Returns the DXF string for all vertex components.

dxftags() → Iterable[ezdxf.llldxf.types.DXFTag]
Returns all vertex components as single DXFTag objects.

NONE_TAG
ezdxf.llldxf.types.NONE_TAG
Special tag representing a none existing tag.

Tags
A list of DXFTag, inherits from Python standard list. Unlike the statement in the DXF Reference “Do not write programs that rely on the order given here”, tag order is sometimes essential and some group codes may appear multiples times in one entity. At the worst case (Material: normal map shares group codes with diffuse map) using same group codes with different meanings.
```python
class e zdxf.lldxf.tags.Tags

Subclass of list.

Collection of DXFTag as flat list. Low level tag container, only required for advanced stuff.

classmethod from_text (text: str) → Tags

Constructor from DXF string.

dxf_type () → str

Returns DXF type of entity, e.g. 'LINE'.

get_handle () → str

Get DXF handle. Raises DXFValueError if handle not exist.

    Returns handle as plain hex string like 'FF00'

    Raises DXFValueError – no handle found

replace_handle (new_handle: str) → None

Replace existing handle.

    Parameters new_handle – new handle as plain hex string e.g. 'FF00'

has_tag (code: int) → bool

Returns True if a DXFTag with given group code is present.

    Parameters code – group code as int

has_embedded_objects () → bool

get_first_tag (code: int, default=DXFValueError) → DXFTag

Returns first DXFTag with given group code or default, if default ! = DXFValueError, else raises DXFValueError.

    Parameters

        • code – group code as int

        • default – return value for default case or raises DXFValueError

get_first_value (code: int, default=DXFValueError) → Any

Returns value of first DXFTag with given group code or default if default != DXFValueError, else raises DXFValueError.

    Parameters

        • code – group code as int

        • default – return value for default case or raises DXFValueError

find_all (code: int) → List[DXFTag]

Returns a list of DXFTag with given group code.

    Parameters code – group code as int

filter (codes: Iterable[int]) → Iterable[DXFTag]

Iterate and filter tags by group codes.

    Parameters codes – group codes to filter

collect_consecutive_tags (codes: Iterable[int], start: int = 0, end: int = None) → Tags

Collect all consecutive tags with group code in codes, start and end delimits the search range. A tag code not in codes ends the process.

    Parameters

        • codes – iterable of group codes
```
• **start** – start index as int
• **end** – end index as int, `None` for end index = `len(self)`

**Returns** collected tags as `Tags`

**tag_index** *(code: int, start: int = 0, end: int = None) → int*

Return index of first `DXFTag` with given group code.

**Parameters**
• **code** – group code as int
• **start** – start index as int
• **end** – end index as int, `None` for end index = `len(self)`

**update** *(tag: DXFTag)*

Update first existing tag with same group code as `tag`, raises `DXFValueError` if `tag` not exist.

**set_first** *(tag: DXFTag)*

Update first existing tag with group code `tag.code` or append `tag`.

**remove_tags** *(codes: Iterable[int]) → None*

Remove all tags inplace with group codes specified in `codes`.

**Parameters**
• **codes** – iterable of group codes as int

**remove_tags_except** *(codes: Iterable[int]) → None*

Remove all tags inplace except those with group codes specified in `codes`.

**Parameters**
• **codes** – iterable of group codes

**pop_tags** *(codes: Iterable[int]) → Iterable[DXFTag]*

Pop tags with group codes specified in `codes`.

**Parameters**
• **codes** – iterable of group codes

**classmethod strip** *(tags: Tags, codes: Iterable[int]) → Tags*

Constructor from `tags`, strips all tags with group codes in `codes` from tags.

**Parameters**
• **tags** – iterable of `DXFTag`
• **codes** – iterable of group codes as int

**ezdxf.lldxf.tags.group_tags** *(tags: Iterable[DXFTag], splitcode: int = 0) → Iterable[Tags]*

Group of tags starts with a SplitTag and ends before the next SplitTag. A SplitTag is a tag with code == `splitcode`, like `(0, ‘SECTION’)` for `splitcode` == 0.

**Parameters**
• **tags** – iterable of `DXFTag`
• **splitcode** – group code of split tag
xdata
    XDATA as list of Tags

embedded_objects
    embedded objects as list of Tags

noclass
    Short cut to access first subclass.

get_handle () → str
    Returns handle as hex string.

dxf$type () → str
    Returns DXF type as string like “LINE”.

replace_handle (handle: str) → None
    Replace the existing entity handle by a new value.

legacy_repair ()
    Legacy (DXF R12) tags handling and repair.

clon,e () → ExtendedTags
    Shallow copy.

flatten_subclasses ()
    Flatten subclasses in legacy mode (DXF R12).
    There exists DXF R12 with subclass markers, technical incorrect but works if the reader ignore subclass
    marker tags, unfortunately ezdxf tries to use this subclass markers and therefore R12 parsing by ezdxf does
    not work without removing these subclass markers.
    This method removes all subclass markers and flattens all subclasses into ExtendedTags.noclass.

get subclass (name: str, pos: int = 0) → Tags
    Get subclass name.

    Parameters
    • name – subclass name as string like “AcDbEntity”
    • pos – start searching at subclass pos.

has_xdata (appid: str) → bool
    True if has XDATA for appid.

get_xdata (appid: str) → Tags
    Returns XDATA for appid as Tags.

set_xdata (appid: str, tags: IterableTags) → None
    Set tags as XDATA for appid.

new_xdata (appid: str, tags: 'IterableTags' = None) → Tags
    Append a new XDATA block.
    Assumes that no XDATA block with the same appid already exist:

    try:
        xdata = tags.get_xdata('EZDXF')
    except ValueError:
        xdata = tags.new_xdata('EZDXF')

has_app_data (appid: str) → bool
    True if has application defined data for appid.
get_app_data (appid: str) → Tags
Returns application defined data for appid as Tags including marker tags.

get_app_data_content (appid: str) → Tags
Returns application defined data for appid as Tags without first and last marker tag.

set_app_data_content (appid: str, tags: IterableTags) → None
Set application defined data for appid for already exiting data.

new_app_data (appid: str, tags: 'IterableTags' = None, subclass_name: str = None) → Tags
Append a new application defined data to subclass subclass_name.
Assumes that no app data block with the same appid already exist:

```python
try:
    app_data = tags.get_app_data('{ACAD_REACTORS', tags)
except ValueError:
    app_data = tags.new_app_data('{ACAD_REACTORS', tags)
```

classmethod from_text (text: str, legacy: bool = False) → ExtendedTags
Create ExtendedTags from DXF text.

Packed DXF Tags

Store DXF tags in compact data structures as list or array.array to reduce memory usage.

class ezdxf.lldxf.packedtags.TagList (data: Iterable = None)
Store data in a standard Python list.

Args:
data: iterable of DXF tag values.

data
    Data storage as list.

clear () → TagList
Returns a deep copy.

classmethod from_tags (tags: Tags, code: int) → TagList
Setup list from iterable tags.

Parameters
    • tags – tag collection as Tags
    • code – group code to collect

clear () → None
Delete all data values.

class ezdxf.lldxf.packedtags.TagArray (data: Iterable = None)
TagArray is a subclass of TagList, which store data in an array.array. Array type is defined by class variable DTYPE.

Args:
data: iterable of DXF tag values.

DTYPE
    array.array type as string

values
    Data storage as array.array

set_values (values: Iterable[T_co]) → None
Replace data by values.
class `ezdxf.lldxf.packedtags.VertexArray(data: Iterable = None)`

Store vertices in an `array.array('d')`. Vertex size is defined by class variable `VERTEX_SIZE`.

**Args:**
- `data`: iterable of vertex values as linear list e.g. `[x1, y1, x2, y2, x3, y3, ...]`.

**VERTEX_SIZE**
Size of vertex (2 or 3 axis).

__len__() → int
Count of vertices.

__getitem__(index: int) → Vertex
Get vertex at `index`, extended slicing supported.

__setitem__(index: int, point: Sequence[float]) → None
Set vertex point at `index`, extended slicing not supported.

__delitem__(index: int) → None
Delete vertex at `index`, extended slicing supported.

__iter__() → Iterable[Sequence[float]]
Returns iterable of vertices.

__str__() → str
String representation.

**insert**(pos: int, point: Sequence[float])
Insert `point` in front of vertex at index `pos`.

**Parameters**
- `pos` – insert position
- `point` – point as tuple

append(point: Sequence[float]) → None
Append `point`.

extend(points: Iterable[Sequence[float]]) → None
Extend array by `points`.

set(points: Iterable[Sequence[float]]) → None
Replace all vertices by `points`.

clear() → None
Delete all vertices.

clon() → VertexArray
Returns a deep copy.

classmethod from_tags(tags: Iterable[DXFVertex], code: int = 10) → VertexArray
Setup point array from iterable tags.

**Parameters**
- `tags` – iterable of `DXFVertex`
- `code` – group code to collect

export_dxf(tagwriter: ezdxf.lldxf.tagwriter.TagWriter, code=10)
6.12.4 Documentation Guide

Formatting Guide

This section is only for me, because of the long pauses between develop iterations, I often forget to be consistent in documentation formatting.

Documentation is written with Sphinx and reSturcturedText.

Started integration of documentation into source code and using autodoc features of Sphinx wherever useful.

Sphinx theme provided by Read the Docs:

```
pip install sphinx-rtd-theme
```

guide — Example module

guide.example_func(a:int, b:str, test:str=None, flag:bool=True) → None

Parameters $a$ and $b$ are positional arguments, argument $test$ defaults to $None$ and $flag$ to $True$. Set $a$ to 70 and $b$ to “x” as an example. Inline code examples `example_func(70, 'x')` or simple `example_func(70, "x")`

- arguments: $a$, $b$, $test$ and $flags$
- literal number values: 1, 2... 999
- literal string values: “a String”
- literal tags: (5, “F000”)
- inline code: call a `example_func(x)`
- Python keywords: `None`, `True`, `False`, `tuple`, `list`, `dict`, `str`, `int`, `float`
- Exception classes: `DXFAttributeError`

class guide.ExampleCls(**kwargs)

The `ExampleCls` constructor accepts a number of optional keyword arguments. Each keyword argument corresponds to an instance attribute, so for example

```
e = ExampleCls(flag=True)
```

flag

This is the attribute $flag$.

set_axis(axis)

axis as (x, y, z) tuple

Args:
axis: (x, y, z) tuple

example_method(flag:bool=False) → None

Method `example_method()` of class `ExampleCls`

Text Formatting

DXF version

DXF R12 (AC1009), DXF R2004 (AC1018)

DXF Types

DXF types are always written in uppercase letters but without further formatting: DXF, LINE, CIRCLE

(internal API)
Marks methods as internal API, gets no public documentation.
(internal class) Marks classes only for internal usage, gets not public documentation.

Spatial Dimensions 2D and 3D with an uppercase letter D

Axis x-axis, y-axis and z-axis

Planes xy-plane, xz-plane, yz-plane

Layouts modelspace, paperspace [layout], block [layout]

Extended Entity Data AppData, XDATA, embedded object, APPID

6.13 Glossary

ACI AutoCAD Color Index (ACI)

ACIS The 3D ACIS Modeler (ACIS) is a geometric modeling kernel developed by Spatial Corp. ® (formerly Spatial Technology), part of Dassault Systems.

bulge The Bulge value is used to create arc shaped line segments in Polyline and LWPolyline entities.

CAD Computer-Assisted Drafting or Computer-Aided Design

CTB Color dependent plot style table (ColorDependentPlotStyles)

DWG Proprietary file format of AutoCAD ®. Documentation for this format is available from the Open Design Alliance (ODA) at their Downloads section. This documentation is created by reverse engineering therefore not perfect nor complete.

DXF Drawing eXchange Format is a file format used by AutoCAD ® to interchange data with other CAD applications. DXF is a trademark of Autodesk ®. See also What is DXF?

STB Named plot style table (NamedPlotStyles)

ture color RGB color representation, a combination red, green and blue values to define a color.

6.14 Indices and tables

• genindex
• search
ezdxf, 137
ezdxf.addons, 386
ezdxf.addons.acadctb, 417
ezdxf.addons.drawing, 387
ezdxf.addons.dxf2code, 398
ezdxf.addons.geo, 390
ezdxf.addons.importer, 395
ezdxf.addons.iterdxf, 400
ezdxf.addons.odafc, 409
ezdxf.addons.pycsg, 411
ezdxf.addons.r12writer, 403
ezdxf.addons.text2path, 410
ezdxf.bbox, 361
ezdxf.comments, 367
ezdxf.disassemble, 359
ezdxf.document, 140
ezdxf.entities, 210
ezdxf.entities.dxfflags, 209
ezdxf.entitydb, 520
ezdxf.layouts, 187
ezdxf.lldxf.extendedtags, 525
ezdxf.lldxf.packedtags, 527
ezdxf.lldxf.tags, 523
ezdxf.lldxf.types, 522
ezdxf.math, 299
ezdxf.options, 364
ezdxf.path, 347
ezdxf.query, 296
ezdxf.recover, 146
ezdxf.render, 369
ezdxf.render.forms, 374
ezdxf.render.point, 386
ezdxf.render.trace, 384
ezdxf.reorder, 369
ezdxf.sections.blocks, 153
ezdxf.sections.classes, 151
ezdxf.sections.entities, 154
ezdxf.sections.header, 150
ezdxf.sections.objects, 154
ezdxf.sections.table, 157
ezdxf.sections.tables, 152
ezdxf.tools.fonts, 517
ezdxf.units, 27
ezdxf.zoom, 366
guide, 529
Symbols

_index_() (ezdxf.math.Vec3 method), 319
__add__() (ezdxf.addons.pycsg.CSG method), 416
__add__() (ezdxf.math.Matrix method), 345
__add__() (ezdxf.math.Vec3 method), 320
__bool__() (ezdxf.math.Vec3 method), 320
__contains__() (ezdxf.entities.Dictionary method), 287
__contains__() (ezdxf.entities.dxffirst.DXFGroup method), 209
__contains__() (ezdxf.entities.dxffirst.GroupCollection method), 210
__contains__() (ezdxf.entitydb.entitydb.EntityDB method), 520
__contains__() (ezdxf.layouts.BlockLayout method), 208
__contains__() (ezdxf.layouts.Layout method), 204
__contains__() (ezdxf.layouts.Layouts method), 187
__contains__() (ezdxf.sections.blocks.BlocksSection method), 153
__contains__() (ezdxf.sections.header.HeaderSection method), 150
__contains__() (ezdxf.sections.objects.ObjectsSection method), 155
__contains__() (ezdxf.sections.tables.Table method), 157
__copy__() (ezdxf.math.Matrix44 method), 315
__copy__() (ezdxf.math.Vec3 method), 319
__deepcopy__() (ezdxf.math.Vec3 method), 319
__delitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 419
__delitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 418
__delitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 520
__delitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 528
__delitem__() (ezdxf.sections.blocks.BlocksSection method), 153
__delitem__() (ezdxf.sections.header.HeaderSection method), 150
__eq__() (ezdxf.addons.types.DXFTag method), 522
__eq__() (ezdxf.math.Matrix method), 345
__eq__() (ezdxf.math.Vec3 method), 320
__geo_interface__ (ezdxf.addons.geo.GeoProxy attribute), 392
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 253
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 520
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 419
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 418
__getitem__() (ezdxf.addons.DXFTag method), 287
__getitem__() (ezdxf.addons.DimStyleOverride method), 225
__getitem__() (ezdxf.addons.DimStyleOverride method), 224
__getitem__() (ezdxf.addons.MLineStyleElements method), 253
__getitem__() (ezdxf.addons.MLineStyleElements method), 252
__getitem__() (ezdxf.addons.MLineStyleElements method), 251
__getitem__() (ezdxf.addons.acadctb.BaseLayout method), 189
__getitem__() (ezdxf.addons.acadctb.EntityDB method), 520
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 521
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 520
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 521
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 520
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 521
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 520
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 521
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 520
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 521
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 520
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 521
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 520
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 521
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 520
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 521
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 520
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 521
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 520
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 521
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 520
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 521
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 520
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 521
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 520
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 521
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 520
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 521
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 520
__getitem__() (ezdxf.addons.acadctb.EntitySpace method), 521
ezdxf Documentation, Release 0.16.2

add_extruded_surface()
  (ezdxf.layouts.BaseLayout method), 203
add_face()  (ezdxf.entities.MeshData method), 254
add_face()  (ezdxf.render.MeshBuilder method), 380
add_foreign_entity()  (ezdxf.layouts.BaseLayout method), 190
add_geodata()  (ezdxf.sections.objects.ObjectsSection method), 155
add_hatch()  (ezdxf.layouts.BaseLayout method), 197
add_image()  (ezdxf.layouts.BaseLayout method), 197
add_image_def()  (ezdxf.document.Drawing method), 144
add_image_def()  (ezdxf.sections.objects.ObjectsSection method), 156
add_import()  (ezdxf.addons.dxf2code.Code method), 399
add_leader()  (ezdxf.layouts.BaseLayout method), 202
add_line()  (ezdxf.addons.dxf2code.Code method), 400
add_polyline()  (ezdxf.addons.r12writer.R12FastStreamWriter method), 407
add_polyline2d()  (ezdxf.layouts.BaseLayout method), 193
add_polyline3d()  (ezdxf.layouts.BaseLayout method), 193
add_polyline_2d()  (ezdxf.addons.r12writer.R12FastStreamWriter method), 407
add_polyline_path()  (ezdxf.entities.BoundaryPaths method), 236
add_polymesh()  (ezdxf.addons.r12writer.R12FastStreamWriter method), 408
add_polymesh()  (ezdxf.layouts.BaseLayout method), 193
add_radius_dim()  (ezdxf.layouts.BaseLayout method), 199
add_radius_dim_2p()  (ezdxf.layouts.BaseLayout method), 200
add_radius_dim_cra()  (ezdxf.layouts.BaseLayout method), 201
add_region()  (ezdxf.layouts.BaseLayout method), 197
add_lines()  (ezdxf.addons.dxf2code.Code method), 400
add_lofted_surface()  (ezdxf.layouts.BaseLayout method), 190
add_mesh()  (ezdxf.layouts.BaseLayout method), 197
add_mesh()  (ezdxf.render.MeshBuilder method), 380
add_mline()  (ezdxf.layouts.BaseLayout method), 195
add_mtext()  (ezdxf.layouts.BaseLayout method), 194
add_multi_point_linear_dim()  (ezdxf.layouts.BaseLayout method), 198
add_new_dict()  (ezdxf.entities.Dictionary method), 287
add_open_spline()  (ezdxf.layouts.BaseLayout method), 196
add_placeholder()  (ezdxf.sections.objects.ObjectsSection method), 156
add_point()  (ezdxf.addons.r12writer.R12FastStreamWriter method), 406
add_point()  (ezdxf.layouts.BaseLayout method), 190
add_polyface()  (ezdxf.addons.r12writer.R12FastStreamWriter method), 407
add_polyface()  (ezdxf.layouts.BaseLayout method), 194
add_polyface()  (ezdxf.addons.r12writer.R12FastStreamWriter method), 407
add_polyface()  (ezdxf.layouts.BaseLayout method), 192
add_solid()  (ezdxf.addons.dxf2code.Code method), 406
add_solid()  (ezdxf.layouts.BaseLayout method), 194
add_spline()  (ezdxf.entities.EdgePath method), 238
add_spline()  (ezdxf.layouts.BaseLayout method), 195
add_spline()  (in module ezdxf.path), 355
add_spline_control_frame()  (ezdxf.layouts.BaseLayout method), 196
add_sweep_surface()  (ezdxf.layouts.BaseLayout method), 204
add_swept_surface()  (ezdxf.layouts.BaseLayout method), 204
add_text()  (ezdxf.addons.dxf2code.Code method), 406
add_text()  (ezdxf.layouts.BaseLayout method), 192
add_text()  (ezdxf.math.ConstructionArc method), 329
add_to_layout()  (ezdxf.math.ConstructionArc method), 329
apply_construction_tool() (ezdxf.entities.Ellipse method), 230
apply_construction_tool() (ezdxf.entities.Spline method), 271
apply_factor (ezdxf.addons.acadctb.ColorDependentPlotStyles attribute), 418
apply_factor (ezdxf.addons.acadctb.NamedPlotStyles attribute), 274
approximate() (ezdxf.math.Bezier method), 337
approximate() (ezdxf.math.EulerSpiral method), 340
approximate() (ezdxf.math.Patch method), 358
approximate() (ezdxf.math.RevolvedSurface method), 339
approximated_length() (ezdxf.math.Bezier3P method), 339
approximated_length() (ezdxf.math.Bezier4P method), 338
Arc (class in ezdxf.entities), 218
arc_angle_span_deg() (in module ezdxf.math), 300
arc_approximation() (ezdxf.math.BSpline static method), 336
arc_center (ezdxf.entities.ArcDimension.dxf attribute), 229
arc_chord_length() (in module ezdxf.math), 302
arc_edges_to_ellipse_edges() (ezdxf.entities.BoundaryPaths method), 236
arc_length_parameterization (ezdxf.entities.LoftedSurface.dxf attribute), 274
arc_segment_count() (in module ezdxf.math), 302
arc_to_bulge() (in module ezdxf.math), 301
ArcDimension (class in ezdxf.entities), 229
ArcEdge (class in ezdxf.entities), 239
area() (in module ezdxf.math), 300
ascending() (in module ezdxf.reorder), 369
aspect_ratio (ezdxf.entities.Viewport.dxf attribute), 174
associate() (ezdxf.entities.Hatch method), 235
associative (ezdxf.entities.Hatch.dxf attribute), 232
attachment_point (ezdxf.entities.Dimension.dxf attribute), 223
attachment_point (ezdxf.entities.MText.dxf attribute), 255
Attdef (class in ezdxf.entities), 186
attribs() (ezdxf.layouts.BlockLayout method), 208
Attrib (class in ezdxf.entities), 185
attribs() (ezdxf.entities.Insert attribute), 182
audit() (ezdxf.document.Drawing method), 145
audit() (ezdxf.entities.dxfgroups.DXFGroup method), 210
audit() (ezdxf.entities.dxfgroups.GroupCollection method), 210
AUTOMATIC (in module ezdxf.addons.acadctb), 421
axis_point (ezdxf.entities.RevolvedSurface.dxf attribute), 274
axis_rotate() (ezdxf.math.Matrix44 method), 316
axis_vector (ezdxf.entities.RevolvedSurface.dxf attribute), 274
B
back_clip_plane_z_value (ezdxf.entities.Viewport.dxf attribute), 282
back_clipping (ezdxf.entities.View.dxf attribute), 175
back_clipping (ezdxf.entities.Viewport.dxf attribute), 174
background_handle (ezdxf.entities.View.dxf attribute), 176
background_handle (ezdxf.entities.Viewport.dxf attribute), 284
banded_matrix() (in module ezdxf.math), 343
BandedMatrixLU (class in ezdxf.math), 346
bank (ezdxf.entities.ExtrudedSurface.dxf attribute), 273
bank (ezdxf.entities.SweptSurface.dxf attribute), 275
base_point (ezdxf.entities.Block.dxf attribute), 180
base_point (ezdxf.entities.PatternLine attribute), 241
base_point_set (ezdxf.entities.ExtrudedSurface.dxf attribute), 273
base_point_set (ezdxf.entities.SweptSurface.dxf attribute), 275
base_ucs_handle (ezdxf.entities.View.dxf attribute), 176
BaseLayout (class in ezdxf.layouts), 188
cased (ezdxf.math.EulerSpiral method), 306
basic_transformation() (in module ezdxf.math), 306
bbox() (in module ezdxf.path), 354
best_fit_normal() (in module ezdxf.math), 306
Bezier (class in ezdxf.math), 337
Bezier (class in ezdxf.render), 372
Bezier3P (class in ezdxf.math), 339
Bezier4P (class in ezdxf.math), 338
bezier_decomposition() (ezdxf.math.BSpline method), 336
bezier_to_bspline() (in module ezdxf.math), 311
BezierSurface (class in ezdxf.math), 340
bg_fill (ezdxf.entities.MText.dxf attribute), 256
bg_fill_color (ezdxf.entities.MText.dxf attribute), 256
bg_fill_color_name (ezdxf.entities.MText.dxf attribute), 256
bg_fill_true_color (ezdxf.entities.MText.dxf attribute), 256

538
blend_crease (ezdxf.entities.Mesh.dxf attribute), 254
Block (class in ezdxf.entities), 180
block (ezdxf.layouts.BlockLayout attribute), 208
block() (ezdxf.entities.Insert method), 182
block_color (ezdxf.entities.Leader.dxf attribute), 245
block_record_handle (ezdxf.entities.GeoData.dxf attribute), 289
block_records (ezdxf.sections.tables.TablesSection attribute), 153
block_to_code() (in module ezdxf.addons.dxf2code), 399
BlockLayout (class in ezdxf.layouts), 208
BlockRecord (class in ezdxf.entities), 178
blocks (ezdxf.addons.dxf2code.Code attribute), 399
blocks (ezdxf.document.Drawing attribute), 141
BlocksSection (class in ezdxf.sections.blocks), 153
Body (class in ezdxf.entities), 219
border_lines() (ezdxf.math.ConstructionBox method), 333
boundary_path (ezdxf.entities.Image attribute), 243
boundary_path (ezdxf.entities.Underlay attribute), 280
boundary_path_wcs() (ezdxf.entities.Image method), 243
BoundaryPaths (class in ezdxf.entities), 236
bounding_box (ezdxf.math.ConstructionArc attribute), 328
bounding_box (ezdxf.math.ConstructionBox attribute), 332
bounding_box (ezdxf.math.ConstructionCircle attribute), 326
bounding_box (ezdxf.math.ConstructionLine attribute), 325
bounding_box (ezdxf.math.Shape2d attribute), 333
BoundingBox (class in ezdxf.math), 323
BoundingBox2d (class in ezdxf.math), 323
box() (in module ezdxf.render.forms), 375
box_fill_scale (ezdxf.entities.MText.dxf attribute), 256
brightness (ezdxf.entities.Image.dxf attribute), 242
BSpline (class in ezdxf.math), 334
bspline() (ezdxf.math.EulerSpiral method), 341
build_system_font_cache() (in module ezdxf.tools.fonts), 519
bulge, 530
bulge (ezdxf.entities.Vertex.dxf attribute), 264
bulge_3_points() (in module ezdxf.math), 302
bulge_center() (in module ezdxf.math), 301
bulge_radius() (in module ezdxf.math), 301
bulge_to_arc() (in module ezdxf.math), 301
bytes_to_hexstr() (in module ezdxf.tools), 368

C

Cache (class in ezdxf.bbox), 364
calendardate() (in module ezdxf.tools), 368
camera_plottable (ezdxf.entities.View.dxf attribute), 176
can_explode (ezdxf.layouts.BlockLayout attribute), 208
cap_height (ezdxf.tools.fonts.FontMeasurements attribute), 519
ccw (ezdxf.entities.ArcEdge attribute), 239
ccw (ezdxf.entities.EllipseEdge attribute), 240
center (ezdxf.entities.Arc.dxf attribute), 218
center (ezdxf.entities.ArcEdge attribute), 239
center (ezdxf.entities.Circle.dxf attribute), 220
center (ezdxf.entities.Ellipse.dxf attribute), 229
center (ezdxf.entities.Viewport.dxf attribute), 281
center (ezdxf.entities.VPort.dxf attribute), 174
center (ezdxf.math.ConstructionArc attribute), 328
center (ezdxf.math.ConstructionBox attribute), 332
center (ezdxf.math.ConstructionCircle attribute), 326
center (ezdxf.math.ConstructionEllipse attribute), 330
center() (in module ezdxf.zoom), 366
center_point (ezdxf.entities.View.dxf attribute), 175
centered (ezdxf.entities.Gradient attribute), 241
chain() (ezdxf.math.Matrix44 static method), 316
chain_layouts_and_blocks() (ezdxf.document.Drawing method), 145
char_height (ezdxf.entities.MText.dxf attribute), 255
Circle (class in ezdxf.entities), 220
circle() (in module ezdxf.render.forms), 375
circle_center() (ezdxf.math.EulerSpiral method), 340
circle_zoom (ezdxf.entities.Viewport.dxf attribute), 282
circle_zoom (ezdxf.entities.VPort.dxf attribute), 174
circumcircle_radius (ezdxf.math.ConstructionBox attribute), 332
class_id (ezdxf.entities.ExtrudedSurface.dxf attribute), 273
class_id (ezdxf.entities.RevolvedSurface.dxf attribute), 274
class_version (ezdxf.entities.ImageDef.dxf attribute), 292
class_version (ezdxf.entities.ImageDefReactor.dxf attribute), 292
classes (ezdxf.document.Drawing attribute), 141
classes (ezdxf.sections.classes.ClassesSection attribute), 151
ClassesSection (class in ezdxf.sections.classes), 151
clear() (ezdxf.entities.GeometryDrawings method), 237
clear() (ezdxf.entities.Dictionary method), 287
clear() (ezdxf.entities.dxffgroups.DXFGroup method), 209
clear() (ezdxf.entities.dxffgroups.GroupCollection method), 210
clear() (ezdxf.entities.EdgedPath method), 238
clear() (ezdxf.entities.LWPolyline method), 249
clear() (ezdxf.entities.MLine method), 251
clear() (ezdxf.entities.Pattern method), 240
clear() (ezdxf.entities.Polyl inePath method), 237
clear() (ezdxf.entitydb.EntitySpace method), 521
clear() (ezdxf.lldxf.packedtags.TagList method), 527
clear() (ezdxf.lldxf.packedtags.VertexArray method), 528

clip_mode (ezdxf.entities.Image.dxf attribute), 243
clip_polygon_2d() (in module ezdxf.math), 304
clipping (ezdxf.entities.Image.dxf attribute), 242
clipping (ezdxf.entities.Underlay attribute), 280
clipping_boundary_handle (ezdxf.entities.Viewport.dxf attribute), 283
clipping_boundary_type (ezdxf.entities.Image.dxf attribute), 243
clockwise() (ezdxf.path.Path method), 358
close() (ezdxf.lldxf.extendedtags.ClosedTags method), 526
close() (ezdxf.lldxf.packedtags.TagList method), 527
clone() (ezdxf.lldxf.packedtags.VertexArray method), 528
clone() (ezdxf.lldxf.types.DXFTag method), 523
color (ezdxf.math.Matrix method), 344
color (ezdxf.math.DXFTag method), 524
color (ezdxf.entities.ezdxf.entities.mline.MLineStyleElement attribute), 253
color (ezdxf.entities.Layer attribute), 160
color (ezdxf.entities.Layer.dxf attribute), 160
color (ezdxf.entities.Sun.dxf attribute), 293
color1 (ezdxf.entities.Gradient attribute), 241
color2 (ezdxf.entities.Gradient attribute), 241
color_name (ezdxf.entities.DXFFormat.dxf attribute), 216
ColorDependentPlotStyles (ezdxf.addons.acadctb), 418
cols() (ezdxf.math.Matrix method), 344
column_count (ezdxf.entities.Insert.dxf attribute), 182
column_spacing (ezdxf.entities.Insert.dxf attribute), 182
columns() (ezdxf.math.Matrix column method), 317
commit() (ezdxf.entities.DimStyleOverride method), 225
compact_banded_matrix() (in module ezdxf.math), 343
cone() (in module ezdxf.render.forms), 378
cone_2p() (in module ezdxf.render.forms), 378
const_width (ezdxf.entities.LWPolyline.dxf attribute), 248
construction_tool() (ezdxf.entities.Arc method), 219
construction_tool() (ezdxf.entities.Ellipse method), 230
construction_tool() (ezdxf.entities.Spline method), 271
ConstructionArc (in ezdxf.math), 327
ConstructionBox (in ezdxf.math), 332
ConstructionCircle (in ezdxf.math), 326
ConstructionEllipse (in ezdxf.math), 330
ConstructionLine (in ezdxf.math), 325
ConstructionRay (in ezdxf.math), 324
contrast (ezdxf.entities.Image.dxf attribute), 242
contrast (ezdxf.entities.Underlay.dxf attribute), 280
close() (ezdxf.entities.SplineEdge method), 271
control_point_count() (ezdxf.entities.Spline method), 271
close() (ezdxf.entities.Spline attribute), 271
close_surfaces (ezdxf.entities.LoftedSurface.dxf attribute), 274
closed_surface() (in module ezdxf.math), 310
closest_point() (in module ezdxf.math), 299
check() (ezdxf.addons.dxf2code.Code method), 399
check() (ezdxf.addons.dxf2code.Code attribute), 399
check() (ezdxf.addons.dxf2code.Code method), 399
col() (ezdxf.math.Matrix method), 344
color (ezdxf.math.DXFTag method), 524
col() (ezdxf.math.Matrix method), 344
context() (ezdxf.addons.dxf2code.Code method), 399
col() (ezdxf.math.Matrix method), 344
context() (ezdxf.addons.dxf2code.Code attribute), 399
col() (ezdxf.math. MLineStyleElement attribute), 253
col() (ezdxf.math.Matrix method), 344
context() (ezdxf.addons.dxf2code.Code attribute), 399
col() (ezdxf.math.Matrix method), 344
context() (ezdxf.addons.dxf2code.Code method), 399
col() (ezdxf.math. MLineStyleElement attribute), 253
context() (ezdxf.addons.dxf2code.Code attribute), 399
col() (ezdxf.math.Matrix method), 344
context() (ezdxf.addons.dxf2code.Code method), 399
col() (ezdxf.math. MLineStyleElement attribute), 253
context() (ezdxf.addons.dxf2code.Code attribute), 399
col() (ezdxf.math.Matrix method), 344
context() (ezdxf.addons.dxf2code.Code attribute), 399
col() (ezdxf.math.Matrix method), 344
context() (ezdxf.addons.dxf2code.Code method), 399
col() (ezdxf.math. MLineStyleElement attribute), 253
context() (ezdxf.addons.dxf2code.Code attribute), 399
col() (ezdxf.math.Matrix method), 344
context() (ezdxf.addons.dxf2code.Code attribute), 399
col() (ezdxf.math.Matrix method), 344
context() (ezdxf.addons.dxf2code.Code method), 399
Index
delete_attrb() (ezdxf.entities.Insert method), 184
delete_block() (ezdxf.sections.blocks.BlockSection method), 153
delete_config() (ezdxf.sections.table.ViewportTable method), 159
delete_entity() (ezdxf.layouts.BaseLayout method), 189
delete_layout() (ezdxf.document.Drawing method), 144
derivative() (ezdxf.math.Bezier method), 337
derivative() (ezdxf.math.BSpline method), 335
derivatives() (ezdxf.math.Bezier method), 338
derivatives() (ezdxf.math.BSpline method), 335
descender_height (ezdxf.tools.fonts.FontMeasurements attribute), 519
descending() (in module ezdxf.reorder), 369
description (ezdxf.addons.acadctb.ColorDependentPlotStyles attribute), 418
description (ezdxf.addons.acadctb.NamedPlotStyles attribute), 419
description (ezdxf.addons.acadctb.PlotStyle attribute), 420
description (ezdxf.entities.dxfgroups.DXFGroup.dxf attribute), 209
description (ezdxf.entities.Layer attribute), 161
description (ezdxf.entities.Linetype.dxf attribute), 165
description (ezdxf.entities.MLineStyle.dxf attribute), 252
design_point (ezdxf.entities.GeoData.dxf attribute), 289
detect_banded_matrix() (in module ezdxf.math), 343
determinant() (ezdxf.math.BandedMatrixLU method), 347
determinant() (ezdxf.math.LU Decomposition method), 346
determinant() (ezdxf.math.Matrix method), 345
DgnDefinition (class in ezdxf.entities), 295
DgnUnderlay (class in ezdxf.entities), 281
diag() (ezdxf.math.Matrix method), 344
Dictionary (class in ezdxf.entities), 286
DictionaryWithDefault (class in ezdxf.entities), 288
dimadec (ezdxf.entities.DimStyle.dxf attribute), 168
dimalt (ezdxf.entities.DimStyle.dxf attribute), 168
dimald (ezdxf.entities.DimStyle.dxf attribute), 168
dimaltf (ezdxf.entities.DimStyle.dxf attribute), 168
dimalltrnd (ezdxf.entities.DimStyle.dxf attribute), 167
dimalttdd (ezdxf.entities.DimStyle.dxf attribute), 168
dimalttzz (ezdxf.entities.DimStyle.dxf attribute), 169
dimaltu (ezdxf.entities.DimStyle.dxf attribute), 168
dimaltz (ezdxf.entities.DimStyle.dxf attribute), 169
dimapost (ezdxf.entities.DimStyle.dxf attribute), 166
dimasz (ezdxf.entities.DimStyle.dxf attribute), 166
dimatfit (ezdxf.entities.DimStyle.dxf attribute), 170
dimaunit (ezdxf.entities.DimStyle.dxf attribute), 169
dimazin (ezdxf.entities.DimStyle.dxf attribute), 168
dimblk (ezdxf.entities.DimStyle.dxf attribute), 166
dimblk1 (ezdxf.entities.DimStyle.dxf attribute), 166
dimblk1_handle (ezdxf.entities.DimStyle.dxf attribute), 170
dimblk2 (ezdxf.entities.DimStyle.dxf attribute), 166
dimblk2_handle (ezdxf.entities.DimStyle.dxf attribute), 170
dimblk_handle (ezdxf.entities.DimStyle.dxf attribute), 170
dimcen (ezdxf.entities.DimStyle.dxf attribute), 167
dimclrd (ezdxf.entities.DimStyle.dxf attribute), 168
dimclrt (ezdxf.entities.DimStyle.dxf attribute), 168
dimdec (ezdxf.entities.DimStyle.dxf attribute), 168
dimdle (ezdxf.entities.DimStyle.dxf attribute), 166
dimdli (ezdxf.entities.DimStyle.dxf attribute), 166
dimdsep (ezdxf.entities.DimStyle.dxf attribute), 169
Dimension (class in ezdxf.entities), 222
dimension (ezdxf.entities.DimStyleOverride attribute), 225
dimex (ezdxf.entities.DimStyle.dxf attribute), 166
dimexo (ezdxf.entities.DimStyle.dxf attribute), 166
dimfit (ezdxf.entities.DimStyle.dxf attribute), 169
dimfrac (ezdxf.entities.DimStyle.dxf attribute), 169
dimfxl (ezdxf.entities.DimStyle.dxf attribute), 170
dimfxlon (ezdxf.entities.DimStyle.dxf attribute), 170
dimgap (ezdxf.entities.DimStyle.dxf attribute), 167
dimjust (ezdxf.entities.DimStyle.dxf attribute), 169
dimlblrblk (ezdxf.entities.DimStyle.dxf attribute), 170
dimlblrblk_handle (ezdxf.entities.DimStyle.dxf attribute), 170
dilex1l_handle (ezdxf.entities.DimStyle.dxf attribute), 170
dilex2l_handle (ezdxf.entities.DimStyle.dxf attribute), 170
dimlfac (ezdxf.entities.DimStyle.dxf attribute), 167
dimlim (ezdxf.entities.DimStyle.dxf attribute), 167
dimlt (ezdxf.entities.DimStyle.dfx attribute), 170
dimltx2 (ezdxf.entities.DimStyle.dxf attribute), 170
dimlttype (ezdxf.entities.DimStyle.dxf attribute), 170
dimlttype_handle (ezdxf.entities.DimStyle.dxf attribute), 170
dimlunit (ezdxf.entities.DimStyle.dxf attribute), 167
dimlwd (ezdxf.entities.DimStyle.dxf attribute), 170
dimlwe (ezdxf.entities.DimStyle.dxf attribute), 170
dimpost (ezdxf.entities.DimStyle.dxf attribute), 166
dimrnd (ezdxf.entities.DimStyle.dxf attribute), 166
dimsah (ezdxf.entities.DimStyle.dxf attribute), 168
dimscale (ezdxf.entities.DimStyle.dxf attribute), 166
dxf (ezdxf.entities.DXFEntity attribute), 211
dxf (ezdxf.layouts.BlockLayout attribute), 208
dfx (ezdxf.layouts.Layer attribute), 204
dfx_entities () (in module ezdxf.addons.geo), 392
dxfattrs () (ezdxf.entities.DXFEntity method), 212
dxfattrs () (ezdxf.math.ConstructionEllipse method), 331
DXFBinaryTag (class in ezdxf.lldxf.types), 523
DXFClass (class in ezdxf.entities), 151
DXFEntity (class in ezdxf.entities), 211
DXFGraphic (class in ezdxf.entities), 214
DXFGroup (class in ezdxf.entities.dxfgroups), 209
DXFLayout (class in ezdxf.entities), 292
DXFObject (class in ezdxf.entities), 286
dxfstr () (ezdxf.lldxf.types.DXFBinaryTag method), 523
dxfstr () (ezdxf.lldxf.types.DXFTag method), 523
dxfstr () (ezdxf.lldxf.types.DXFVertex method), 523
DXFTag (class in ezdxf.lldxf.types), 522
dxf_tag () (in module ezdxf.lldxf.types), 522
dxftags () (ezdxf.lldxf.types.DXFVertex method), 523
dxf_type () (ezdxf.entities.DXFEntity method), 211
dxf_type () (ezdxf.lldxf.extendedtags.ExtendedTags method), 256
dxfversion () (ezdxf.entities.MeshData method), 140
DXFVertex (class in ezdxf.lldxf.types), 523

e
edge_create_values (ezdxf.entities.MeshData attribute), 254
EdgePath (class in ezdxf.entities), 237
generates (ezdxf.entities.MeshData attribute), 254
generates (ezdxf.entities.MeshData attribute), 254
generates (ezdxf.render.MeshBuilder attribute), 380
generates_as_vertices () (ezdxf.render.MeshBuilder method), 380
edit_data () (ezdxf.entities.dxfgroups.DXFGroup method), 209
edit_data () (ezdxf.entities.Mesh method), 254
elements (ezdxf.entities.ezdxf.entities.mline.MLineStyleElements attribute), 253
elements (ezdxf.entities.MLineStyle attribute), 252
elevation (ezdxf.entities.Hatch attribute), 232
elevation (ezdxf.entities.LWPolyline attribute), 248
elevation (ezdxf.entities.Polyline attribute), 261
elevation (ezdxf.entities.View attribute), 176
elevation (ezdxf.entities.Viewport attribute), 284
Ellipse (class in ezdxf.entities), 229
ellipse () (in module ezdxf.render.forms), 375
ellipse_approximation () (ezdxf.math.BSpline static method), 336
ellipse_edges_to_spline_edges () (ezdxf.entities.BoundaryPaths method), 236
ellipse_param_span () (in module ezdxf.math), 301
EllipseEdge (class in ezdxf.entities), 239
eLLipticTransformation () (in module ezdxf.path), 356
embedded_objects (ezdxf.lldxf.extendedtags.ExtendedTags attribute), 526
encode () (ezdxf.document.Drawing method), 143
encode () (in module ezdxf.tools.crypt), 368
encode_base64 () (ezdxf.document.Drawing method), 143
end (ezdxf.entities.Line.dxf attribute), 246
end (ezdxf.entities.LineEdge attribute), 239
end (ezdxf.math.ConstructionEllipse attribute), 330
end (ezdxf.math.ConstructionLine attribute), 325
end (ezdxf.path.Path attribute), 357
end_angle (ezdxf.entities.Arc.dxf attribute), 218
end_angle (ezdxf.entities.ArcDimension attribute), 229
end_angle (ezdxf.entities.ArcEdge attribute), 239
end_angle (ezdxf.entities.EllipseEdge attribute), 240
end_angle (ezdxf.entities.MLineStyle attribute), 252
end_angle (ezdxf.math.ConstructionArc attribute), 328
end_angle_rad (ezdxf.math.ConstructionArc attribute), 328
end_draft_angle (ezdxf.entities.LoftedSurface.dxf attribute), 274
end_draft_distance (ezdxf.entities.RevolvedSurface.dxf attribute), 275
end_draft_magnitude (ezdxf.entities.LoftedSurface attribute), 274
end_param (ezdxf.entities.Ellipse.dxf attribute), 230
end_point (ezdxf.entities.Arc dxf attribute), 219
end_point (ezdxf.entities.Ellipse dxf attribute), 230
end_point (ezdxf.math.ConstructionArc attribute), 328
end_point (ezdxf.math.ConstructionEllipse attribute), 330
end_style (ezdxf.addons.acad.dxfPlotStyle attribute), 420
end_tangent (ezdxf.entities.Spline.dxf attribute), 271
end_tangent (ezdxf.entities.SplineEdge attribute), 240
end_width (ezdxf.entities.Vertex.dxf attribute), 264
EndBlk (class in ezdxf.entities), 181
dndBlk (ezdxf.layouts.BlockLayout attribute), 208
entities (ezdxf.document.Drawing attribute), 141
ezdxf.recover (module), 146
ezdxf.render (module), 369
ezdxf.render.forms (module), 374
ezdxf.render.point (module), 386
ezdxf.render.trace (module), 384
ezdxf.reorder (module), 369
ezdxf.sections.blocks (module), 153
ezdxf.sections.classes (module), 151
ezdxf.sections.entities (module), 154
ezdxf.sections.header (module), 150
ezdxf.sections.objects (module), 154
ezdxf.sections.table (module), 157
ezdxf.sections.tables (module), 152
ezdxf.tools.fonts (module), 517
ezdxf.units (module), 27
ezdxf.zoom (module), 366
F
Face3d (class in ezdxf.entities), 217
faces (ezdxf.entities.GeoData attribute), 290
faces (ezdxf.entities.Mesh attribute), 254
faces (ezdxf.entities.MeshData attribute), 254
faces (ezdxf.entities.MeshBuilder attribute), 380
faces (ezdxf.entities.MeshBuilderPolyface method), 267
faces (ezdxf.entities.MeshBuilderTrace.CurvedTrace method), 385
faces (ezdxf.entities.MeshBuilderTrace.LinearTrace method), 385
faces (ezdxf.entities.MeshBuilderTrace.TraceBuilder method), 384
faces_as_vertices (ezdxf.entities.MeshBuilder method), 380
fade (ezdxf.entities.Image.dxf attribute), 243
fade (ezdxf.entities.Underlay.dxf attribute), 280
family (ezdxf.tools.fonts.FontFace attribute), 519
fast_zoom (ezdxf.entities.VPort.dxf attribute), 174
field_length (ezdxf.entities.Attdef.dxf attribute), 186
filename (ezdxf.document.Drawing attribute), 141
filename (ezdxf.entities.ImageDef.dxf attribute), 292
filename (ezdxf.entities.UnderlayDefinition.dxf attribute), 294
fill_color (ezdxf.entities.MLineStyle.dxf attribute), 252
fill_params (ezdxf.entities.MLineVertex attribute), 252
fill_style (ezdxf.addons.acad2d.PlotStyle attribute), 420
filter (ezdxf.addons.geo.GeoProxy method), 394
filter (ezdxf.lldxf.tags.Tags method), 524
filter_invalid_xdata_group_codes (module ezdxf.options), 365
finalize (ezdxf.addons.importer.Importer method), 396
find_all (ezdxf.lldxf.tags.Tags method), 524
find_shx (ezdxf.sections.table.StyleTable method), 158
first (ezdxf.query.EntityQuery attribute), 297
fit_length (ezdxf.entities.Text method), 278
fit_pathas_into_box (module ezdxf.path), 354
fit_point_count (ezdxf.entities.Spline method), 271
fit_points (ezdxf.entities.Spline attribute), 271
fit_points (ezdxf.entities.SplineEdge attribute), 240
fit_points_to_cad_cv (module ezdxf.math), 307
fit_points_to_cubic_bezier (module ezdxf.math), 308
fit_tolerance (ezdxf.entities.Spline.dxf attribute), 270
flag (guide.ExampleCls attribute), 529
flags (ezdxf.entities.AppID.dxf attribute), 177
flags (ezdxf.entities.Block.dxf attribute), 180
flags (ezdxf.entities.Body.dxf attribute), 220
flags (ezdxf.entities.DimStyle.dxff attribute), 166
flags (ezdxf.entities.DXFTable.dxff attribute), 152
flags (ezdxf.entities.Image.dxff attribute), 242
flags (ezdxf.entities.Layer.dxff attribute), 159
flags (ezdxf.entities.LWPolyline.dxff attribute), 248
flags (ezdxf.entities.MLine.dxff attribute), 251
flags (ezdxf.entities.MLineStyle.dxff attribute), 252
flags (ezdxf.entities.Polyline.dxff attribute), 261
flags (ezdxf.entities.Spline.dxff attribute), 270
flags (ezdxf.entities.Textstyle.dxff attribute), 164
flags (ezdxf.entities.UCS.dxff attribute), 177
flags (ezdxf.entities.Underlay.dxff attribute), 279
flags (ezdxf.entities.Vertex.dxff attribute), 264
flags (ezdxf.entities.View.dxff attribute), 175
flags (ezdxf.entities.Viewport.dxff attribute), 282
flags (ezdxf.entities.VPort.dxff attribute), 173
flatten_subclasses (ezdxf.lldxf.extendedtags.ExtendedTags method), 526
flattening (ezdxf.entities.Arc method), 219
flattening (ezdxf.entities.Circle method), 220
flattening (ezdxf.entities.Ellipse method), 230
flattening (ezdxf.entities.Spline method), 271
flattening (ezdxf.math.Bezier method), 337
flattening (ezdxf.math.Bezier3P method), 339
flattening (ezdxf.math.Bezier4P method), 338
flattening (ezdxf.math.BSpline method), 335
flattening (ezdxf.math.ConstructionEllipse method), 331
flattening (ezdxf.path.Path method), 358
float2transparency (module ezdxf), 367
flow_direction (ezdxf.entities.MText.dxff attribute), 255
font (ezdxf.entities.Textstyle.dxff attribute), 278
font_cache_directory (module ezdxf.options), 364
font_name (ezdxf.entities.Text method), 278
FontFace (class in ezdxf.tools.fonts), 518
get_table_lineweight()
    (ezdxf.addons.acadctb.ColorDependentPlotStyles
     method), 418
get_table_lineweight()
    (ezdxf.addons.acadctb.NamedPlotStyles
     method), 419
get_underlay_def()
    (ezdxf.entities.Underlay
     method), 280
global_bspline_interpolation() (in module
ezdxf.math), 308
globe_to_map() (ezdxf.addons.geo.GeoProxy
     method), 394
Gradient (class in ezdxf.entities), 241
gradient (ezdxf.entities.Hatch attribute), 232
graphic_properties()
    (ezdxf.entities.DXFGraphic
     method), 215
grayscale (ezdxf.addons.acadctb.PlotStyle
     attribute), 421
grid() (ezdxf.entities.Insert method), 183
grid_frequency (ezdxf.entities.Viewport.dxf
     attribute), 284
grid_on (ezdxf.entities.VPort.dxf attribute), 174
grid_spacing (ezdxf.entities.Viewport.dxf
     attribute), 282
grid_spacing (ezdxf.entities.VPort.dxf
     attribute), 174
group_tags() (in module ezdxf.lldxf.tags), 525
groupby() (ezdxf.document.Drawing
     method), 143
groupby() (ezdxf.layouts.BaseLayout
     method), 189
groupby() (ezdxf.query.EntityQuery
     method), 298
GroupCollection (class in ezdxf.entities.dxffroups),
    210
groups (ezdxf.document.Drawing
     attribute), 141
groups() (ezdxf.entities.dxffroups.GroupCollection
     method), 210
guid() (in module ezdxf.tools), 368
guide (module), 529
H
halign (ezdxf.entities.Text.dxf attribute), 276
handle (ezdxf.entities.Block.dxf attribute), 180
handle (ezdxf.entities.DXFEntity.dxf attribute), 211
handle (ezdxf.entities.EndBlk.dxf attribute), 181
handle (ezdxf.entities.Layer.dxf attribute), 159
handle (ezdxf.entities.Textstyle.dxf attribute), 164
handles() (ezdxf.entities.dxffroups.DXFGroup
     method), 209
hard_owned (ezdxf.entities.Dictionary.dxf
     attribute), 286
has_app_data() (ezdxf.entities.DXFEntity
     method), 212
has_app_data() (ezdxf.lldxf.extendedtags.ExtendedTags
     method), 526
has_attr() (ezdxf.entities.Polyline
     attribute), 262
has_binary_data (ezdxf.entities.Body
     attribute), 220
has_clockwise_orientation() (ezdxf.path.Path
     method), 358
has_curves (ezdxf.path.Path
     attribute), 358
has_dxf_attrib() (ezdxf.entities.DXFEntity
     method), 358
has_dxf_unicode() (in module ezdxf), 367
has_embedded_objects() (ezdxf.lldxf.tags.Tags
     method), 524
has_entry() (ezdxf.sections.table.Table
     method), 157
has_extension_dict (ezdxf.entities.DXFEntity
     attribute), 212
has_gradient_data (ezdxf.entities.Hatch
     attribute), 233
has_handle() (ezdxf.entitydb.EntitySpace
     method), 521
has_hookline (ezdxf.entities.Leader.dxf
     attribute), 244
has_hyperlink() (ezdxf.entities.DXFGraphic
     method), 215
has_intersection() (ezdxf.math.ConstructionLine
     method), 326
has_leader (ezdxf.entities.ArcDimension.dxf
     attribute), 229
has_lines (ezdxf.path.Path
     attribute), 358
has_none_planar_faces() (ezdxf.render.MeshBuilder
     method), 380
has_pattern_fill (ezdxf.entities.Hatch
     attribute), 233
has_reactors() (ezdxf.entities.DXFEntity
     method), 213
has_scaling (ezdxf.entities.Insert
     attribute), 182
has_solid_fill (ezdxf.entities.Hatch
     attribute), 233
has_tag() (ezdxf.lldxf.tags.Tags
     method), 524
has_tag() (ezdxf.sections.header.CustomVars
     method), 151
has_uniform_scaling (ezdxf.entities.Insert
     attribute), 182
has_width (ezdxf.entities.Polyline
     attribute), 262
has_xdata() (ezdxf.entities.DXFEntity
     method), 212
has_xdata() (ezdxf.lldxf.extendedtags.ExtendedTags
     method), 526
import_entity()
import_modelspace()
import_entities()
import_blocks()
import_block()
ImageDefReactor
ImageDef
image_size
image_handle
image_def_handle
Image
image_def
identity()
id
history_handle
hits
hookline_direction
horizontal_direction
horizontal_unit_scale
horizontal_units
id
identity()
Image
image_def
image_def_handle
image_handle
image_size
image_size
ImageDef
ImageDefReactor
import_block()
import_blocks()
import_entities()
import_entity()
import_modelspace()
intersect_circle()
(324)
intersect_ray()
(327)
intersection_line_line_2d() (in module ezdxf.math), 303
intersection_ray_ray_3d() (in module ezdxf.math), 307
invalidate() (ezdxf.bbox.Cache method), 364
invisible (ezdxf.entities.DXFGraphic.dxf attribute), 216
invisible_edge (ezdxf.entities.Face3d.dxf attribute), 217
is_2d_polyline (ezdxf.entities.Polyline attribute), 262
is_2d_polyline_vertex (ezdxf.entities.Vertex attribute), 265
is_3d_polyline (ezdxf.entities.Polyline attribute), 262
is_3d_polyline_vertex (ezdxf.entities.Vertex attribute), 265
is_active_paperspace (ezdxf.entities.BlockRecord attribute), 179
is_active_paperspace (ezdxflayouts.BaseLayout attribute), 188
is_alive (ezdxf.layouts.BaseLayout attribute), 188
is_an_entity (ezdxf.entities.DXFClass.dxf attribute), 152
is_anonymous (ezdxf.entities.Block attribute), 181
is_any_corner_inside()
(ezdxf.math.ConstructionBox method), 333
is_any_layout (ezdxf.entities.BlockRecord attribute), 179
is_any_layout (ezdxf.layouts.BaseLayout attribute), 189
is_anym_paperspace (ezdxf.entities.BlockRecord attribute), 179
is_anym_paperspace (ezdxf.layouts.BaseLayout attribute), 188
is_block_layout (ezdxf.entities.BlockRecord attribute), 179
is_block_layout (ezdxflayouts.BaseLayout attribute), 189
is_cartesian (ezdxf.math.UCS attribute), 312
is_close_points() (in module ezdxf.math), 299
is_coplanar (ezdxf.math.Plane method), 322
is_coplanar_vertex() (ezdxf.math.Plane method), 322
is_face_record (ezdxf.entities.Vertex attribute), 265
is_frozen() (ezdxf.entities.Layer method), 161
is_locked() (ezdxf.entities.Layer attribute), 161
is_m_closed (ezdxf.entities.Polyline attribute), 262
is_modelspace (ezdxf.entities.BlockRecord attribute), 179
is_modelspace (ezdxf.layouts.BaseLayout attribute), 189
is_n_closed (ezdxf.entities.Polyline attribute), 262
is_null (ezdxf.math.Vec3 attribute), 319
is_off() (ezdxf.entities.Layer method), 161
is_on() (ezdxf.entities.Layer method), 161
is_overlapping() (ezdxf.math.ConstructionBox method), 333
is_parallel() (ezdxf.math.ConstructionRay method), 324
is_parallel() (ezdxf.math.Vec3 method), 320
is_partial (ezdxf.entities.ArcDimension.dxf attribute), 229
is_planar_face() (in module ezdxf.math), 306
is_point_in_polygon_2d() (in module ezdxf.math), 303
is_point_left_of_line()
(ezdxf.math.ConstructionLine method), 326
is_point_left_of_line() (in module ezdxf.math), 303
is_point_on_line_2d() (in module ezdxf.math), 303
is_poly_face_mesh (ezdxf.entities.Polyline attribute), 262
is_poly_face_mesh_vertex (ezdxf.entities.Vertex attribute), 265
is_polygon_mesh (ezdxf.entities.Polyline attribute), 262
is_polygon_mesh_vertex (ezdxf.entities.Vertex at-
lineweight (ezdxf.entities.Pattern attribute), 420
lines (ezdxf.entities.Pattern attribute), 240
Linetype (class in ezdxf.entities), 165
linetype (ezdxf.entities.acadctb.PlotStyle attribute), 420
linetype (ezdxf.entities.DXFGraphic.dxf attribute), 216
linetype (ezdxf.entities.ezdxf_entities.mline.MLineStyleElement attribute), 253
linetype (ezdxf.entities.Layer.dxf attribute), 160
linetypes (ezdxf.entities.acadctb.Code attribute), 399
linetypes (ezdxf.document.Drawing attribute), 142
linetypes (ezdxf.sections.tables.tables section attribute), 152
lineweight (ezdxf.entities.acadctb.PlotStyle attribute), 420
lineweight (ezdxf.entities.DXFGraphic.dxf attribute), 216
lineweight (ezdxf.entities.Layer.dxf attribute), 160
lineweights (ezdxf.entities.acadctb.ColorDependentPlotStyles attribute), 418
lineweights (ezdxf.entities.acadctb.NamedNamedPlotStyles attribute), 419
linspace() (module ezdxf.math), 300
list() (ezdxf.math.Vec3 class method), 320
live_selection_handle (ezdxf.entities.View.dxf attribute), 176
load() (module ezdxf.entities.acadctb), 417
load() (module ezdxf.tools.fonts), 520
load_proxy_graphics (module ezdxf.options), 365
loaded (ezdxf.entities.ImageDef.dxf attribute), 292
local_cubic_bezier_interpolation() (module ezdxf.math), 309
location (ezdxf.entities.MLineVertex attribute), 252
location (ezdxf.entities.Point.dxf attribute), 260
location (ezdxf.entities.Vertex.dxf attribute), 264
location (ezdxf.math.ConstructionRay attribute), 324
lock() (ezdxf.entities.Layer method), 161
LoftedSurface (class in ezdxf.entities), 274
log_unprocessed_tags (module ezdxf.options), 365
lower (ezdxf.math.BandedMatrixLU attribute), 346
lower_left (ezdxf.entities.VPort.dxf attribute), 173
lt_scale (ezdxf.entities.DXFGraphic.dxf attribute), 216
LU_decomp() (ezdxf.math.Matrix method), 345
LU Decomposition (class in ezdxf.math), 345
luminance() (module ezdxf.colors), 368
LPolyline (class in ezdxf.entities), 248
M
m1 (ezdxf.math.BandedMatrixLU attribute), 346
m2 (ezdxf.math.BandedMatrixLU attribute), 346
m_close() (ezdxf.entities.Polyline method), 263
m_count (ezdxf.entities.Polyline.dxf attribute), 261
m_smooth_density (ezdxf.entities.Polyline.dxf attribute), 262
magnification (ezdxf.math.Vec3 attribute), 319
magnitude (ezdxf.math.Vec3 attribute), 319
magnitude_square (ezdxf.math.Vec3 attribute), 319
magnitude_xy (ezdxf.math.Vec3 attribute), 319
Element_axis_points() (ezdxf.math.ConstructionEllipse method), 331
main_viewport() (ezdxf.layouts.Paperspace method), 207
major_axis (ezdxf.entities.Ellipse.dxf attribute), 229
major_axis (ezdxf.math.ConstructionEllipse attribute), 330
major_axis_vector (ezdxf.entities.EllipseEdge attribute), 239
make_font() (module ezdxf.tools.fonts), 517
make_path() (module ezdxf.path), 348
make_primitive() (module ezdxf.disassemble), 360
map_to_globe() (ezdxf.addons.geo.GeoProxy method), 394
material_handle (ezdxf.entities.Layer.dxf attribute), 160
materials (ezdxf.document.Drawing attribute), 142
MatplotlibFont (class in ezdxf.tools.fonts), 518
Matrix (class in ezdxf.math), 343
Matrix44 (class in ezdxf.math), 314
max_flattening_distance (ezdxf.disassemble.Primitive attribute), 361
mcount (ezdxf.entities.Insert attribute), 182
MengerSponge (class in ezdxf.addons), 426
merge() (ezdxf.entities.dxf2code.Code method), 399
Mesh (class in ezdxf.entities), 253
mesh() (ezdxf.addons.MengerSponge method), 426
mesh() (ezdxf.addons.pycsg.CSG method), 416
mesh() (ezdxf.addons.SierpinskyPyramid method), 431
mesh_faces_count (ezdxf.entities GeoData.dxf attribute), 290
MeshAverageVertexMerger (class in ezdxf.render), 383
MeshBuilder (class in ezdxf.render), 379
MeshData (class in ezdxf.entities), 254
MeshTransformer (class in ezdxf.render), 382
MeshVertexCache (class in ezdxf.entities), 266
MeshVertexMerger (class in ezdxf.render), 383
midpoint() (ezdxf.math.ConstructionLine method), 325
minor_axis (ezdxf.entities.Ellipse attribute), 230
minor_axis (ezdxf.math.ConstructionEllipse attribute), 330
minor_axis_length (ezdxf.entities.EllipseEdge attribute), 240

Index 553
nullvec (ezdxf.math.Plane attribute), 322
null_vector (ezdxf.entities.Leader.dxf attribute), 245
null_vector_3p() (in module ezdxf.math), 306
normalize() (ezdxf.math.Vec3 method), 320
normalize_text_angle() (in module ezdxf.tools), 368
north_direction (ezdxf.entities.GeoData.dxf attribute), 289
nrows (ezdxf.math.BandedMatrixLUGeo attribute), 346
nrows (ezdxf.math.BezierSurface attribute), 340
nrows (ezdxf.math.LUDecomposition attribute), 346
nrows (ezdxf.math.Matrix attribute), 344
NULLVEC (in module ezdxf.math), 321

O
OBJECT_COLOR (in module ezdxf.addons.acadctb), 421
OBJECT_COLOR2 (in module ezdxf.addons.acadctb), 421
OBJECT_LINETYPE (in module ezdxf.addons.acadctb), 421
OBJECT_LINEWEIGHT (in module ezdxf.addons.acadctb), 421
objects (ezdxf.document.Drawing attribute), 141
objects() (in module ezdxf.zoom), 366
ObjectsSection (class in ezdxf.sections.objects), 155
oblique (ezdxf.entities.Shape.dxf attribute), 268
oblique (ezdxf.entities.Text.dxf attribute), 276
oblique (ezdxf.entities.Textstyle.dxf attribute), 164
oblique_angle (ezdxf.entities.Dimension.dxf attribute), 224
observation_from_tag (ezdxf.entities.GeoData.dxf attribute), 290
observation_to_tag (ezdxf.entities.GeoData.dxf attribute), 290
OCS (class in ezdxf.math), 311
cos() (ezdxf.entities.DXFGraphic method), 214
cos() (ezdxf.entities.Layer method), 161
cos() (ezdxf.entities.line.mline.MLineStyleElement method), 253
cos() (ezdxf.entities.PatternLine attribute), 241
cos() (ezdxf.math.Shape2d method), 334
cos() (ezdxf.math.Use) (in module ezdxf.math), 304
on (ezdxf.entities.VideoCapture attribute), 280
on() (ezdxf.entities.Layer method), 161
one_color (ezdxf.entities.Gradient attribute), 241
open_uniform_bspline() (in module ezdxf.math), 310
open_uniform_knot_vector() (in module ezdxf.math), 299
opendxf() (in module ezdxf.addons.iterdxf), 401
optimize() (ezdxf.entities.MeshData method), 254
optimize() (ezdxf.entities.Polyface method), 267

origin (ezdxf.entities.UCSTable.dxf attribute), 178
orthogonal() (ezdxf.math.ConstructionRay method), 324
orthogonal() (ezdxf.math.Vec3 method), 320
outermost_paths() (ezdxf.entities.BoundaryPaths method), 236
output_encoding (ezdxf.document.Drawing attribute), 141
override() (ezdxf.entities.Dimension method), 224
owner (ezdxf.entities.AppID.dxf attribute), 177
owner (ezdxf.entities.Block.dxf attribute), 180
owner (ezdxf.entities.BlockRecord.dxf attribute), 178
owner (ezdxf.entities.DimStyle.dxf attribute), 166
owner (ezdxf.entities.DXFEntity.dxf attribute), 211
owner (ezdxf.entities.EndBlk.dxf attribute), 181
owner (ezdxf.entities.Layer.dxf attribute), 159
owner (ezdxf.entities.LineType.dxf attribute), 165
owner (ezdxf.entities.TextStyle.dxf attribute), 164
owner (ezdxf.entities.UCS.dxf attribute), 177
owner (ezdxf.entities.View.dxf attribute), 175
owner (ezdxf.entities.VPort.dxf attribute), 173

P
page_setup() (ezdxf.layouts.Paperspace method), 207
page_setup_name (ezdxf.entities.PlotSettings.dxf attribute), 293
Paperspace (class in ezdxf.layouts), 207
paperspace (ezdxf.entities.DXFGraphic.dxf attribute), 216
params() (ezdxf.entities.Ellipse method), 230
params() (ezdxf.math.Bezier method), 337
params() (ezdxf.math.BSpline method), 335
params() (ezdxf.math.ConstructionEllipse method), 330
params() (ezdxf.math.Use) (in module ezdxf.math), 304
params() (ezdxf.math.Use) (in module ezdxf.math), 304
Path (class in ezdxf.path), 357
path_entity_id (ezdxf.entities.SweptSurface.dxf attribute), 275
path_entity_transform_computed (ezdxf.entities.ExtrudedSurface.dxf attribute), 273
path_entity_transform_computed (ezdxf.entities.SweptSurface.dxf attribute), 275
path_entity_transformation_matrix (ezdxf.entities.ExtrudedSurface.dxf attribute), 273
path_entity_transformation_matrix() (ezdxf.entities.SweptSurface method), 276

Index

555
Index
render_splines_and_polylines() (in module ezdxf.path), 350
render_uniform_bspline() (ezdxf.render.Spline method), 370
render_uniform_rbspline() (ezdxf.render.Spline method), 371
rendering_paths() (ezdxf.entities.BoundaryPaths method), 236
replace() (ezdxf.math.Vec3 method), 319
replace() (ezdxf.sections.header.CustomVars method), 151
replace_handle() (ezdxf.lldxf.extendedtags.ExtendedTags method), 526
replace_xdata_list() (ezdxf.entities.DXFEntity method), 213
required_knot_values() (in module ezdxf.math), 300
reset_boundary_path() (ezdxf.entities.Image method), 243
reset_boundary_path() (ezdxf.entities.Underlay method), 280
reset_extents() (ezdxf.layouts.Layout method), 204
reset_fingerprint_guid() (ezdxf.document.Drawing method), 145
reset_limits() (ezdxf.layouts.Layout method), 204
reset_main_viewport() (ezdxf.layouts.Paperspace method), 207
reset_paper_limits() (ezdxf.layouts.Paperspace method), 208
reset_transformation() (ezdxf.entities.Insert method), 184
reset_version_guid() (ezdxf.document.Drawing method), 145
reset_viewports() (ezdxf.layouts.Paperspace method), 207
reshape() (ezdxf.math.Matrix static method), 344
resolution_units (ezdxf.entities.ImageDef attribute), 292
reverse() (ezdxf.math.Bezier method), 337
reverse() (ezdxf.math.Bezier3P method), 339
reverse() (ezdxf.math.Bezier4P method), 338
reverse() (ezdxf.math.BSpline method), 335
reversed() (ezdxf.math.Vec3 method), 320
reversed() (ezdxf.path.Path method), 358
revolve_angle (ezdxf.entities.RevolvingSurface attribute), 274
RevolvingSurface (class in ezdxf.entities), 274
rgb (ezdxf.entities.DXFGraphic attribute), 214
rgb (ezdxf.entities.Layer attribute), 160
rgb2int() (in module ezdxf), 367
rootdict (ezdxf.sections.objects.ObjectsSection attribute), 155
rootdict (ezdxf.sections.objects.ObjectsSection method), 333
rotate() (ezdxf.math.ConstructionBox method), 333
rotate() (ezdxf.math.Shape2d method), 334
rotate() (ezdxf.math.UCS method), 313
rotate() (ezdxf.math.Vec3 method), 321
rotate_axis() (ezdxf.entities.DXFGraphic method), 215
rotate_axis() (ezdxf.render.MeshTransformer method), 383
rotate_deg() (ezdxf.math.Vec3 method), 321
rotate_local_x() (ezdxf.math.UCS method), 313
rotate_local_y() (ezdxf.math.UCS method), 313
rotate_local_z() (ezdxf.math.Vec3 method), 313
rotate_rad() (ezdxf.math.Shape2d method), 334
rotate_x() (ezdxf.entities.DXFGraphic method), 215
rotate_x() (ezdxf.render.MeshTransformer method), 382
rotate_y() (ezdxf.entities.DXFGraphic method), 215
rotate_y() (ezdxf.render.MeshTransformer method), 382
rotate_z() (ezdxf.entities.DXFGraphic method), 215
rotate_z() (ezdxf.math.ConstructionArc method), 328
rotate_z() (ezdxf.render.MeshTransformer method), 383
rotation (ezdxf.entities.Gradient attribute), 241
rotation (ezdxf.entities.Insert.dxf attribute), 182
rotation (ezdxf.entities.MText.dxf attribute), 256
rotation (ezdxf.entities.Shape.dxf attribute), 268
rotation (ezdxf.entities.Text.dxf attribute), 276
rotation (ezdxf.entities.Underlay.dxf attribute), 279
rotation_form() (in module ezdxf.render.forms), 379
row() (ezdxf.math.Matrix method), 344
row_count (ezdxf.entities.Insert.dxf attribute), 182
row_spacing (ezdxf.entities.Insert.dxf attribute), 182
rows() (ezdxf.math.Matrix method), 344
rows() (ezdxf.math.Matrix44 method), 317
ruled_surface (ezdxf.entities.LoftedSurface attribute), 274
rytz_axis_construction() (in module ezdxf.math), 304
S
save() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 418
save() (ezdxf.addons.acadctb.NamedPlotStyles method), 420
save() (ezdxf.document.Drawing method), 142
save() (in module ezdxf.tools.fonts), 520
saveas() (ezdxf.document.Drawing method), 142
scale (ezdxf.entities.BlockRecord.dxf attribute), 178
scale (ezdxf.entities.Underlay attribute), 280
set_location() (ezdxf.entities.DimStyleOverride method), 226
set_location() (ezdxf.entities.MText method), 228
set_masking_area() (ezdxf.entities.Wipeout method), 285
set_mesh_vertex() (ezdxf.entities.Polymesh method), 266
set_modelspace_vport() (ezdxf.document.Drawing method), 145
set_open_rational() (ezdxf.entities.Spline method), 271
set_open_uniform() (ezdxf.entities.Spline method), 271
set_pattern_angle() (ezdxf.entities.Hatch method), 233
set_pattern_definition() (ezdxf.entities.Hatch method), 233
set_pattern_fill() (ezdxf.entities.Hatch method), 234
set_pattern_scale() (ezdxf.entities.Hatch method), 233
set_plot_flags() (ezdxf.layouts.Layout method), 206
set_plot_style() (ezdxf.layouts.Layout method), 205
set_plot_type() (ezdxf.layouts.Layout method), 205
set_plot_window() (ezdxf.layouts.Layout method), 205
set_points() (ezdxf.entities.LWPolyline method), 249
set_pos() (ezdxf.entities.Text method), 277
set_raster_variables() (ezdxf.document.Drawing method), 144
set_raster_variables() (ezdxf.sections.objects.ObjectsSection method), 156
set_reactors() (ezdxf.entities.DXFEntity method), 213
set_redraw_order() (ezdxf.layouts.Layout method), 205
set_rotation() (ezdxf.entities.MText method), 257
set_row() (ezdxf.math.Matrix method), 344
set_row() (ezdxf.math.Matrix44 method), 315
set_scale() (ezdxf.entities.Insert method), 182
set_scale_factor() (ezdxf.entities.MLine method), 251
set_seed_points() (ezdxf.entities.Hatch method), 235
set_solid_fill() (ezdxf.entities.Hatch method), 234
set_style() (ezdxf.entities.MLine method), 251
set_table_lineweight() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 418
set_table_lineweight() (ezdxf.addons.acadctb.NamedPlotStyles method), 419
set_text() (ezdxf.entities.Body method), 220
set_text() (ezdxf.entities.DimStyleOverride method), 227
set_text_align() (ezdxf.entities.DimStyle method), 171
set_text_align() (ezdxf.entities.DimStyleOverride method), 226
set_text_format() (ezdxf.entities.DimStyle method), 171
set_text_format() (ezdxf.entities.DimStyleOverride method), 226
set_tick() (ezdxf.entities.DimStyle method), 171
set_tick() (ezdxf.entities.DimStyleOverride method), 226
set_tolerance() (ezdxf.entities.DimStyle method), 172
set_uniform() (ezdxf.entities.Spline method), 271
set_uniform_rational() (ezdxf.entities.Spline method), 272
set_values() (ezdxf.lldxf.packedtags.TagArray method), 527
set_vertices() (ezdxf.entities.Leader method), 245
set_vertices() (ezdxf.entities.PolylinePath method), 237
set_wipeout_variables() (ezdxf.document.Drawing method), 144
set_wipeout_variables() (ezdxf.sections.objects.ObjectsSection method), 157
set_xdata() (ezdxf.entities.DXFEntity method), 213
set_xdata() (ezdxf.lldxf.extendedtags.ExtendedTags method), 526
set_xdata_list() (ezdxf.entities.DXFEntity method), 213
setup_local_grid() (ezdxf.entities.GeoData method), 291
shade_plot_handle (ezdxf.entities.Viewport.dxf attribute), 284
shade_plot_mode (ezdxf.entities.Viewport.dxf attribute), 284
shadow_map_size (ezdxf.entities.Sun.dxf attribute), 294

Index
start_location() (ezdxf.entities.MLine method), 251
start_param (ezdxf.entities.Ellipse.dxf attribute), 230
start_point (ezdxf.entities.EllipseArc attribute), 218
start_point (ezdxf.entities.Ellipse attribute), 230
start_point (ezdxf.math.ConstructionArc attribute), 328
start_point (ezdxf.math.ConstructionEllipse attribute), 330
start_tangent (ezdxf.entities.Spline.dxf attribute), 271
start_tangent (ezdxf.entities.SplineEdge attribute), 240
start_width (ezdxf.entities.Vertex.dxf attribute), 264
status (ezdxf.entities.Sun.dxf attribute), 293
status (ezdxf.entities.Viewport.dxf attribute), 281
status (ezdxf.entities.VPort.dxf attribute), 174
STB, 530
store_proxy_graphics (in module ezdxf.options), 365
stretch (ezdxf.tools.fonts.FontFace attribute), 519
strip() (ezdxf.llldxf.tags.Tags class method), 525
style (ezdxf.entities.MText.dxf attribute), 255
style (ezdxf.entities.Text.dxf attribute), 276
style (ezdxf.tools.fonts.FontFace attribute), 519
style_element_count (ezdxf.entities.MLine.dxf attribute), 251
style_handle (ezdxf.entities.MLine.dxf attribute), 250
style_name (ezdxf.entities.MLine.dxf attribute), 250
styles (ezdxf.addons.dxf2code.Code attribute), 399
styles (ezdxf.document.Drawing attribute), 141
styles (ezdxf.sections.tables.TablesSection attribute), 152
StyleTable (class in ezdxf.sections.table), 158
subclasses (ezdxf.llldxf.extendedtags.ExtendedTags attribute), 525
subdivide() (ezdxf.render.MeshTransformer method), 382
subdivide() (ezdxf.render.Spline method), 370
subdivide_face() (in module ezdxf.math), 306
subdivide_ngons() (in module ezdxf.math), 306
subdivision_levels (ezdxf.entities.Mesh.dxf attribute), 254
subtract() (ezdxf.addons.pycsg.CSG method), 416
sum() (ezdxf.math.Vec3 static method), 321
Sun (class in ezdxf.entities), 293
sun_handle (ezdxf.entities.View.dxf attribute), 177
sun_handle (ezdxf.entities.Viewport.dxf attribute), 285
suppress_zeros() (in module ezdxf.tools), 368
Surface (class in ezdxf.entities), 272
swap_axis() (ezdxf.math.ConstructionEllipse method), 331
swap_cols() (ezdxf.math.Matrix method), 345
swap_rows() (ezdxf.math.Matrix method), 345
swap_alignment (ezdxf.entities.SweptSurface.dxf attribute), 275
swap_alignment_flags (ezdxf.entities.ExtrudedSurface.dxf attribute), 273
swap_entity_transform_computed (ezdxf.entities.ExtrudedSurface.dxf attribute), 273
swap_entity_transform_computed (ezdxf.entities.SweptSurface.dxf attribute), 275
swap_entity_transformation_matrix (ezdxf.entities.ExtrudedSurface attribute), 273
swap_entity_transformation_matrix() (ezdxf.entities.SweptSurface method), 276
swap_vector (ezdxf.entities.ExtrudedSurface.dxf attribute), 273
swept_entity_id (ezdxf.entities.SweptSurface.dxf attribute), 275
SweptSurface (class in ezdxf.entities), 275
T
Table (class in ezdxf.sections.table), 157
table_entries_to_code() (in module ezdxf.addons.dxf2code), 399
tables (ezdxf.document.Drawing attribute), 141
TablesSection (class in ezdxf.sections.tables), 152
tag (ezdxf.entities.Attdef.dxf attribute), 186
tag (ezdxf.entities.Attrib.dxf attribute), 185
tag_index() (ezdxf.llldxf.tags.Tags method), 525
TagArray (class in ezdxf.llldxf.packedtags), 527
TagList (class in ezdxf.llldxf.packedtags), 527
tags (class in ezdxf.llldxf.tags), 523
tags (ezdxf.entities.XRecord attribute), 296
tangent (ezdxf.entities.Vertex.dxf attribute), 265
tangent() (ezdxf.math.Bezier3P method), 339
tangent() (ezdxf.math.Bezier4P method), 339
tangent() (ezdxf.math.ConstructionCircle method), 326
tangent() (ezdxf.math.EulerSpiral method), 340
tangents() (ezdxf.math.ConstructionArc method), 328
target_point (ezdxf.entities.View.dxf attribute), 175
target_point (ezdxf.entities.Viewport.dxf attribute), 174
target_vertices (ezdxf.entities.GeoData attribute), 290
temp_path (in module ezdxf.addons.odafc), 410
Text (class in ezdxf.entities), 276
text (ezdxf.entities.Attrib.dxf attribute), 186
text (ezdxf.entities.Attrib.dxf attribute), 186
text (ezdxf.entities.Dimension.dxf attribute), 223
text (ezdxf.entities.MText.dxf attribute), 256
text (ezdxf.entities.Text.dxf attribute), 276

562 Index
text_direction (ezdxf.entities.MText.dxf attribute), 255

text_generation_flag (ezdxf.entities.Text.dxf attribute), 277

text_height (ezdxf.entities.Leader.dxf attribute), 244

text_midpoint (ezdxf.entities.Dimension.dxf attribute), 223

text_rotation (ezdxf.entities.Dimension.dxf attribute), 224

text_width (ezdxf.entities.Leader.dxf attribute), 244

text_width() (ezdxf.tools.fonts MatplotlibFont method), 518

text_width() (ezdxf.tools.fonts MonospaceFont method), 518

TextStyle (class in ezdxf.entities), 164

thaw() (ezdxf.entities.Layer method), 161

thickness (ezdxf.entities.DXFGraphic.dxf attribute), 216

thickness (ezdxf.entities.Line.dxf attribute), 246
time (ezdxf.entities.Sun.dxf attribute), 294
tint (ezdxf.entities.Gradient attribute), 241
to_bsplines_and_vertices() (in module ezdxf.path), 353
to_control_vertices() (in module ezdxf.disassemble), 361
to_dxf_entities() (ezdxf.addons.geo GeoProxy method), 393
to_ellipse() (ezdxf.entities.Arc method), 219
to_ellipse() (ezdxf.entities.Circle method), 221
to_hatches() (in module ezdxf.path), 352
to_lines() (in module ezdxf.path), 352
to_lwpolylines() (in module ezdxf.path), 351
to_matplotliblib_path() (in module ezdxf.path), 353
to_meshes() (in module ezdxf.disassemble), 360
to_occs() (ezdxf.math.ConstructionEllipse method), 330
to_occs() (ezdxf.math.UCS method), 312
to_occs_angle_deg() (ezdxf.math.UCS method), 313
to_paths() (in module ezdxf.disassemble), 360
to_polylines2d() (in module ezdxf.path), 351
to_polylines3d() (in module ezdxf.path), 352
to_primitives() (in module ezdxf.disassemble), 360
to_qpainter_path() (in module ezdxf.path), 353
to_spline() (ezdxf.entities.Arc method), 219
to_spline() (ezdxf.entities.Circle method), 221
to_spline() (ezdxf.entities.Ellipse method), 230
to_splines_and_polylines() (in module ezdxf.path), 352
to_vertices() (in module ezdxf.disassemble), 360
to_wcs() (ezdxf.math.OCS method), 311
to_wcs() (ezdxf.math.UCS method), 312
tobytes() (ezdxf.entities.Body method), 220
tostring() (ezdxf.entities.Body method), 220
tostring() (ezdxf.lldims.types.DXFBinaryTag method), 523

Trace (class in ezdxf.entities), 278

TraceBuilder (class in ezdxf.rendertrace), 384

transform() (ezdxf.entities.Arc method), 219

transform() (ezdxf.entities.Circle method), 221

transform() (ezdxf.entities.Dimension method), 224

transform() (ezdxf.entities.DXFGraphic method), 215

transform() (ezdxf.entities.Ellipse method), 230

transform() (ezdxf.entities.Face3d method), 217

transform() (ezdxf.entities.Hatch method), 235

transform() (ezdxf.entities.Image method), 243

transform() (ezdxf.entities.Insert method), 184

transform() (ezdxf.entities.Leader method), 245

transform() (ezdxf.entities.Line method), 246

transform() (ezdxf.entities.LWPolyline method), 250

transform() (ezdxf.entities.Mesh method), 254

transform() (ezdxf.entities.MLine method), 251

transform() (ezdxf.entities.MText method), 258

transform() (ezdxf.entities.Point method), 260

transform() (ezdxf.entities.Polyline method), 263

transform() (ezdxf.entities.Ray method), 267

transform() (ezdxf.entities.Shape method), 268

transform() (ezdxf.entities.Solid method), 269

transform() (ezdxf.entities.Spline method), 272

transform() (ezdxf.entities.Text method), 278

transform() (ezdxf.entities.Trace method), 279

transform() (ezdxf.entities.XLine method), 286

transform() (ezdxf.math.Bezier method), 337

transform() (ezdxf.math.Bezier3P method), 339

transform() (ezdxf.math.Bezier4P method), 338

transform() (ezdxf.math.BSpline method), 335

transform() (ezdxf.math.ConstructionEllipse method), 331

transform() (ezdxf.math.Matrix44 method), 317

transform() (ezdxf.math.UCS method), 313

transform() (ezdxf.math.UCS method), 313

to_wcs() (ezdxf.math.OCS method), 311
to_wcs() (ezdxf.math.UCS method), 312

tostring() (ezdxf.math.Matrix44 method), 317

tostring() (ezdxf.math.Matrix44 method), 317

transform_direction() (ezdxf.math.Matrix44 method), 317

transform_directions() (ezdxf.math.Matrix44 method), 317

transform_paths() (in module ezdxf.path), 354

transform_paths_to_occs() (in module ezdxf.path), 354

transform_vertices() (ezdxf.math.Matrix44 method), 317

transformation_matrix_extruded_entity (ezdxf.entities.ExtrudedSurface attribute), 273

transformation_matrix_path_entity() (ezdxf.entities.SweptSurface method), 275
unnamed (ezdxf.entities.dxffgroups.DXFGroup.dxf attribute), 209
up_direction (ezdxf.entities.GeoData.dxf attribute), 289
update() (ezdxf.entities.DimStyleOverride method), 225
update() (ezdxf.lli.dxf.Tags.Tags method), 525
update_all() (ezdxf.entities.MLineStyle method), 253
update_dxf_attribs() (ezdxf.entities.DXFEntity method), 212
update_geometry() (ezdxf.entities.MLine method), 251
update_instance_counters() (ezdxf.sections.classes.ClassesSection method), 151
update_paper() (ezdxf.layouts.Layout method), 206
upper (ezdxf.math.BandedMatrixLU attribute), 346
upper_right (ezdxf.entities.VPort.dxf attribute), 174
use_matplotlib (in module ezdxf.options), 364
use_plot_styles() (ezdxf.layouts.Layout method), 206
use_standard_scale() (ezdxflayouts.Layout method), 206
user_location_override() (ezdxf.entities.DimStyleOverride method), 228
user_scale_factor (ezdxf.entities.DimStyleOverride method), 290
ux (ezdxf.math.OCS attribute), 311
ux (ezdxf.math.UCS attribute), 312
uy (ezdxf.math.OCS attribute), 311
uy (ezdxf.math.UCS attribute), 312
uz (ezdxf.math.OCS attribute), 311
uz (ezdxf.math.UCS attribute), 312

V

v_count (ezdxf.entities.Surface.dxf attribute), 272
v_pixel (ezdxf.entities.Image.dxf attribute), 242
validate() (ezdxf.document.Drawing method), 145
valign (ezdxf.entities.Text.dxf attribute), 277
value (ezdxf.entities.dxf attribute), 288
values (ezdxf.lli.dxf.packedtags.TagArray attribute), 527
values (ezdxf.lli.dxf.packedtags.TagList attribute), 527
values() (ezdxf.entitydb.EntityDB method), 521
varnames() (ezdxf.sections.header.HeaderSection method), 150

Vec2 (class in ezdxf.math), 322
vec2 (ezdxf.math.Vec3 attribute), 318
Vec3 (class in ezdxf.math), 318
vector (ezdxf.math.Plane attribute), 322
version (ezdxf.entities.Body.dxf attribute), 220
version (ezdxf.entities.Dimension.dxf attribute), 222
version (ezdxf.entities.GeoData.dxf attribute), 289
version (ezdxf.entities.Mesh.dxf attribute), 254
version (ezdxf.entities.Sun.dxf attribute), 293
Vertex (class in ezdxf.entities), 264
VERTEX_SIZE (ezdxf.lli.packedtags.VertexArray attribute), 528
VertexArray (class in ezdxf.lli.packedtags), 527
vertical_unit_scale (ezdxf.entities.GeoData.dxf attribute), 289
vertical_units (ezdxf.entities.GeoData.dxf attribute), 289
vertices (ezdxf.entities.Leader attribute), 245
vertices (ezdxf.entities.Mesh attribute), 254
vertices (ezdxf.entities.MeshData attribute), 254
vertices (ezdxf.entities.MeshVertexCache attribute), 266
vertices (ezdxf.entities.MLineStyle attribute), 251
vertices (ezdxf.entities.Polyline attribute), 262
vertices (ezdxf.entities.PolylinePath attribute), 237
vertices (ezdxf.math.Shape2d attribute), 333
vertices (ezdxf.render.MeshBuilder attribute), 380
vertices() (ezdxf.disassemble.Primitive method), 361
vertices() (ezdxf.entities.Circle method), 220
vertices() (ezdxf.entities.Ellipse method), 230
vertices() (ezdxf.entities.LWPolyline method), 248
vertices() (ezdxf.entities.Solid method), 269
vertices() (ezdxf.entities.Trace method), 279
vertices() (ezdxf.math.ConstructionArc method), 328
vertices() (ezdxf.math.ConstructionEllipse method), 330
vertices_in_wcs() (ezdxf.entities.LWPolyline method), 249
View (class in ezdxf.entities), 175
view_brightness (ezdxf.entities.Viewport.dxf attribute), 284
view_center_point (ezdxf.entities.Viewport.dxf attribute), 282
view_contrast (ezdxf.entities.Viewport.dxf attribute), 284
view_direction_vector (ezdxf.entities.Viewport.dxf attribute), 282
view_height (ezdxf.entities.Viewport.dxf attribute), 282
view_mode (ezdxf.entities.View.dxf attribute), 175
view_mode (ezdxf.entities.VPort.dxf attribute), 174
view_target_point (ezdxf.entities.Viewport.dxf attribute), 282
view_twist (ezdxf.entities.View.dxf attribute), 175
view_twist (ezdxf.entities.VPort.dxf attribute), 174
view_twist_angle (ezdxf.entities.Viewport.dxf attribute), 282
Viewport (class in ezdxf.entities), 281
viewports (ezdxf.document.Drawing attribute), 142
xscale (ezdxf.entities.Insert.dxf attribute), 182
xscale (ezdxf.entities.Shape.dxf attribute), 268
xy (ezdxf.math.Vec3 attribute), 318
xyz (ezdxf.math.Vec3 attribute), 318
xyz_rotate() (ezdxf.math.Matrix44 class method), 316

Y

y (ezdxf.math.Vec3 attribute), 318
Y_AXIS (in module ezdxf.math), 321
y_rotate() (ezdxf.math.Matrix44 class method), 315
yaxis (ezdxf.entities.UCSTable.dxf attribute), 178
yof() (ezdxf.math.ConstructionRay method), 325
yscale (ezdxf.entities.Insert.dxf attribute), 182

Z

z (ezdxf.math.Vec3 attribute), 318
Z_AXIS (in module ezdxf.math), 321
z_rotate() (ezdxf.math.Matrix44 class method), 315
zoom_to_paper_on_update() (ezdxf.layouts.Layout method), 206
zscale (ezdxf.entities.Insert.dxf attribute), 182