## Contents

1 Included Extensions 3
2 Website 5
3 Documentation 7
4 Questions and Feedback at Google Groups 9

5 Contents 11
  5.1 Introduction ........................................ 11
  5.2 Tutorials ............................................ 12
  5.3 Basic Concepts ...................................... 102
  5.4 Reference ............................................ 108
  5.5 Howto ............................................... 303
  5.6 FAQ .................................................. 309
  5.7 Rendering ........................................... 310
  5.8 Add-ons ............................................. 324
  5.9 DXF Internals ....................................... 362
  5.10 Developer Guides .................................. 423
  5.11 Glossary .......................................... 434
  5.12 Indices and tables ................................. 434

Python Module Index 435

Index 437
Welcome! This is the documentation for ezdxf release 0.13, last updated Jul 05, 2020.

- *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 3.6
- OS independent
- additional required packages: pyparsing
- 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
Contents
• **Drawing / Export Addon** add-on to visualise and convert DXF files to images which can be saved to various formats such as png, pdf and svg

• **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

• **Plot Style Files (CTB/STB)** read/write add-on

• **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: http://github.com/mozman/ezdxf.git
Issue Tracker at GitHub: http://github.com/mozman/ezdxf/issues
Questions and Feedback at Google Groups

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

5.1.1 What is ezdxf

ezdxf is a Python interface to the DXF (drawing interchange file) format developed by Autodesk, it 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 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?

5.1.2 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.

5.1.3 Supported Python Versions

ezdxf requires at least Python 3.6 and will be tested with the latest stable CPython 3 version and the latest stable release of pypy3 during development.
ezdxf is written in pure Python 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.

5.1.4 Supported Operating Systems

ezdxf is OS independent and runs on all platforms which provide an appropriate Python interpreter (>=3.6).

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

5.1.6 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.

5.1.7 License

ezdxf is licensed under the very liberal MIT-License.

5.2 Tutorials

5.2.1 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 ezdxf

doc = ezdxf.readfile("your_dxf_file.dxf")
```

See also:

Drawing 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
# 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)
```

```python
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)
```

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`:

- `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 preceding `!` (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:"
    for entity in entities:
        print(' {}
    print('-'*40)

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':
        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(' {}
    print('-'*40)

To exclude entities from the result container the `key` function should return `None`. The `groupby()` function catches `DXFAttributeError` 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
* `r12writer` documentation

5.2.2 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.

### 5.2.3 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 left off this 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:
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 entity 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.

### 5.2.4 Tutorial for Blocks

#### What are Blocks?

Blocks are collections of DXF entities which can be placed multiply times at different layouts and blocks as references to the block definition. 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 the BlocksSection class and every drawing has only one blocks section: 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'
flag = doc.blocks.new(name='FLAG')

# Add DXF entities to the block 'FLAG'.
flag.add_lwpolyline([(0, 0), (0, 5), (4, 3), (0, 3)])  # the flag symbol as 2D polyline
flag.add_circle((0, 0), .4, dxfattribs={'color': 2})  # mark the base point with a circle
```

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:
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 = {
        'NAME': 'P(%(d)02d)' % (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'
flag_refs = []
for block_ref in anonymous_block_refs:
    # Get the block layout of the anonymous block
    block = doc.blocks.get(block_ref.dxf.name)
    # Find all block references to 'FLAG' in the anonymous block
    flag_refs.extend(block.query('INSERT[ name=="FLAG"]'))

# Evaluation example: collect all flag names.
flag_numbers = [flag.get_attrib_text('NAME') for flag in flag_refs if flag.has_attrib('NAME')]
print(flag_numbers)
```

Exploding Block References

New in version 0.12.

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

New in version 0.12.

If you just want to examine the entities of a block reference use the `virtual_entities()` method. This method 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)}.')
```
5.2.5 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 aa 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. bulge > 0 the curve is on the right side of the vertex connection line, 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='xyb')
msp.add_lwpolyline([(0, 10, 0), (10, 10, .5), (20, 10, 0)], format='xyb')
```

5.2.6 Tutorial for Text

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

```python
import ezdxf

# TEXT is a basic entity and is supported by every DXF version.
# Argument setup=True for adding standard linetypes and text styles.
doc = ezdxf.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')
```

(continues on next page)
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`:

```python
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'}`):
LiberationMono-Italic
LiberationMono-BoldItalic
LiberationMono-Bold
LiberationMono
LiberationSerif-Italic
LiberationSerif-BoldItalic
LiberationSerif-Bold
LiberationSerif
LiberationSans-Italic
LiberationSans-BoldItalic
LiberationSans-Bold
LiberationSans
OpenSansCondensed-Italic
OpenSansCondensed-Light
OpenSansCondensed-Bold
OpenSans-ExtraBoldItalic
OpenSans-ExtraBold
OpenSans-BoldItalic
OpenSans-Bold
OpenSans-SemiBoldItalic
OpenSans-SemiBold
OpenSans-Italic
OpenSans
OpenSans-Light-Italic
OpenSans-Light
STANDARD
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 an `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.

5.2.7 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.

mtext.dxf.width = 60

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)"
red text - green text - blue text

**Stacked text**

MText also supports stacked text:

```plaintext
# the space ' ' in front of 'Lower' anr the ';' behind 'Lower' are necessary
# combined with vertical center alignment
mtext.text = "\A1\SUP\ AUpper^ Lower; - \SSUpper/ Lower;) - \SSUpper# Lower;"
```

```
Upper - Upper/ - Upper /
Lower - Lower - 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 ezdxr provides a method to set all required DXF attributes at once:

```plaintext
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.
5.2.8 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)
```
Closed Spline

A closed spline is continuous closed curve.

```python
msp.add_closed_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:

```
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:

```
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:

```
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:

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

Get data count of real existing data:
5.2.9 Tutorial for Polyface

coming soon...

5.2.10 Tutorial for Mesh

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

```python
import ezdxf

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),
]

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),
    (continues on next page)```
(continued from previous page)

```python
(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")
```

5.2.11 Tutorial for Hatch

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 = ezdxf.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=1)

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

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

```python
import ezdxf
doc = ezdxf.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)

# 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=1)

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

(continues on next page)
(continued from previous page)

```python
# 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=1)
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 = ezdxf.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
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>hatch_style</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>nested - altering filled and unfilled areas</td>
</tr>
<tr>
<td>1</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>

```python
hatch = msp.add_hatch(color=1, dxfattribs={
    'hatch_style': 0,
    # 0 = nested
    })
```

(continues on next page)
This is also the result for all 4 paths and hatch_style set to 2 (ignore).

This is also the result for all 4 paths and hatch_style set to 1 (outer).

This is also the result for all 4 paths and hatch_style set to 0 (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=1, flags=0)

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

```
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=1,
                                 )

# 2. edge path
edge_path = hatch.paths.add_edge_path(flags=16)
edge_path.add_spline(
        control_points=[
                        (126.658105895725, 177.0823706957212),
                        (141.5497003747484, 187.8907860433995),
                        (166.4418941042747, 177.0823706957212),
                        (181.3331933052753, 141.5497003747484),
                        (206.225392506276, 126.658105895725),
                        (221.1166917072767, 141.5497003747484),
                        (246.0088910082773, 177.0823706957212),
                        (221.1166917072767, 177.0823706957212),
                        (206.225392506276, 206.2253925062767),
                        (246.0088910082773, 206.2253925062767),
                        (221.1166917072767, 206.2253925062767),
                        (141.5497003747484, 141.5497003747484),
                        (166.4418941042747, 141.5497003747484),
                        (126.658105895725, 126.658105895725)],
                                 flags=16,
                                 )
```
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 gets 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',
    dxfattribs={'closed': 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

5.2.12 Tutorial for Hatch Pattern Definition

TODO

5.2.13 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")
```

5.2.14 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,
```
# 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")

## 5.2.15 Tutorial for Linetypes

Simple line type example:

```
__ __ __ __ __ __ __ __ __ __ __ __ __ __
```

You can define your own line types. A DXF linetype definition consists of name, description and elements:

```python
elements = [total_pattern_length, elem1, elem2, ...]
```

- **total_pattern_length**  Sum of all linetype elements (absolute values)
- **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:

```python
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:

```python
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:

```python
"""
# 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:

```python
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.

### 5.2.16 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----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----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: 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:

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

The parameters are case insensitive. ] 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:

```python
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',
    })
```

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

### 5.2.17 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 Coordinate Systems introduction please.

For 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.

**Object Coordinate System (OCS)**

The OCS is used to place planar 2D entities in 3D space. 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(
    # You can place the 2D circle in 3D space
    # but you have to convert WCS into OCS
    center=ocs.from_wcs((0, 2, 2)),
    # center in OCS: (0.0, 0.0, 2.82842712474619)
    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 \textit{UCS} class in this example to place a 2D pentagon in 3D space.

```python
# The center of the pentagon should be (0, 2, 2), and the shape is
# rotated around x-axis about 45 degree, to accomplish this I use an
# UCS with z-axis (0, 1, 1) and an x-axis parallel to WCS x-axis.
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
)
# calculating corner points in local (UCS) coordinates
points = [Vector.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
    dxfattribs={
        'elevation': elevation,
        'extrusion': ucs.uz,
        'closed': True,
        'color': 1,
    }
)
```

The following image shows the 2D pentagon in 3D space in AutoCAD \textit{Left}, \textit{Front} and \textit{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 \textit{z}-axis of the UCS and the OCS show the same direction (extrusion direction), and the \textit{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 \textit{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 Vector 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,
    dxfattribs={
        'closed': True,
        'color': 1,
    }
)

# add lines from center to corners
center_wcs = ucs.to_wcs((0, 0.333, 0.333))
for corner in corners_wcs:
    msp.add_line(center_wcs, corner, dxfattribs={'color': 1})

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'})

ucs = UCS(origin=(0, 2, 2), ux=(1, 0, 0), uz=(0, 1, 1))

# calculation of text direction as angle in OCS:
# convert text rotation in degree into a vector in UCS
text_direction = Vector.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
ucs = UCS(origin=(0, 2, 2), ux=(1, 0, 0), uz=(0, 1, 1))
msp.add_arc(
    center=ucs.to_ocs((0, 0)),
    radius=1,
    start_angle=ucs.to_ocs_angle_deg(45),
    end_angle=ucs.to_ocs_angle_deg(270),
    dxfattribs={
        'extrusion': ucs.uz,
        'color': 1,
    },
)
center = ucs.to_wcs((0, 0))
msp.add_line(
    start=center,
    end=ucs.to_wcs(Vector.from_deg_angle(45)),
    dxfattribs={'color': 1},
)
msp.add_line(
    start=center,
    end=ucs.to_wcs(Vector.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: Vector, 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 = Vector(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 = Vector(-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)
5.2.18 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.
ucb = 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)
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
ucs = UCS(origin=(0, 2, 2)).rotate_local_x(math.radians(-45))
```
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 Vector 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,
    dxfattribs={
        'closed': True,
        '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)
```

(continues on next page)
msp.add_line(
    start=CENTER,
    end=Vector.from_deg_angle(START_ANGLE),
    dxfattribs={'color': 6},
).transform(ucs.matrix)

msp.add_line(
    start=CENTER,
    end=Vector.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)
```
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
uc = UCS(origin=(1, 2, 0)).rotate_local_x(math.radians(15))
blockref = msp.add_blockref('CSYS', insert=(0, 0, 0))
blockref.transform(uc.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)
```
5.2.19 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 1m 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>0</th>
<th>Center of dimension line</th>
</tr>
</thead>
<tbody>
<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>

```
msp.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>0</th>
<th>Center, it is possible to adjust the vertical location by <code>dimtvp</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Above</td>
</tr>
<tr>
<td>2</td>
<td>Outside, handled like Above by ezdxr</td>
</tr>
<tr>
<td>3</td>
<td>JIS, handled like Above by ezdxr</td>
</tr>
<tr>
<td>4</td>
<td>Below</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>halign</th>
<th>'left', 'right', 'center', 'above1', 'above2'</th>
</tr>
</thead>
<tbody>
<tr>
<td>valign</td>
<td>'above', 'center', 'below'</td>
</tr>
</tbody>
</table>
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**

Besides 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:

```python
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 Textstyle name.</td>
</tr>
<tr>
<td>dimtxt</td>
<td>Text height in drawing units.</td>
</tr>
<tr>
<td>dimclrt</td>
<td>Measurement text color as AutoCAD Color Index (ACI).</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 AutoCAD Color Index (ACI), if dimtfill is 2</td>
</tr>
</tbody>
</table>

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 these 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 the `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.

```
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>

```
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 `dimltex1` and `dimltex2` 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>dimclre</td>
<td>extension line color as <strong>AutoCAD Color Index (ACI)</strong></td>
</tr>
<tr>
<td>dimltex1</td>
<td>linetype of first extension line</td>
</tr>
<tr>
<td>dimltex2</td>
<td>linetype of second extension line</td>
</tr>
<tr>
<td>dimlwe</td>
<td>line weight of extension line</td>
</tr>
<tr>
<td>dimexe</td>
<td>extension beyond dimension line in drawing units</td>
</tr>
<tr>
<td>dimexo</td>
<td>offset of extension line from measurement point</td>
</tr>
<tr>
<td>dimfxlon</td>
<td>set to 1 to enable fixed length extension line</td>
</tr>
<tr>
<td>dimfxl</td>
<td>length of fixed length extension line in drawing units</td>
</tr>
<tr>
<td>dimsel1</td>
<td>suppress first extension line if 1</td>
</tr>
<tr>
<td>dimse2</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
        'dimltex1': 'DASHED2',
        'dimltex2': '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), p1=(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), p1=(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:**

```python
dim = msp.add_linear_dim(base=(3, 2), p1=(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 the 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()
```

<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.
The tolerances ans limits features are implemented by using the `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 `set_tolerance()` method in `DimStyleOverride` or `DimStyle`.

The attribute `dimtp` defines the upper tolerance value, `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!

```
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()
```
Different upper and lower tolerance values:

```python
dim = msp.add_linear_dim(base=(0, 3), p1=(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>
</tbody>
</table>
DIMVAR | Description
---|---
dimtol | set to 1 to enable tolerances
dimtp | set the maximum (or upper) tolerance limit for dimension text
dimtm | set the minimum (or lower) tolerance limit for dimension text
dimtfac | specifies a scale factor for the text height of limits and tolerance values relative to the dimension text height, as set by dimtxt.
dimtzin | 4 to suppress leading zeros, 8 to suppress trailing zeros or 12 to suppress both, like dimzin for dimension text, see also Text Formatting
dimtolj | set the vertical justification for tolerance values relative to the nominal dimension text.
dimtdec | set the number of decimal places to display in tolerance values

dimlim | set to 1 to enable limits

**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.

```
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.

**5.2.20 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 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 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: DIMSTYLE Table
- Source code file standards.py shows how to create your own DIMSTYLES.
- dimension_radius.py for radius 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>dim-tad</td>
<td>1 to place text vertical above the dimension line</td>
</tr>
</tbody>
</table>
ezdxf Documentation, Release 0.13

```python
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 `dimtoh` to 1:

```python
dim = msp.add_radius_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>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmove</td>
<td>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.</td>
</tr>
<tr>
<td>dimtix</td>
<td>1 to force text inside</td>
</tr>
<tr>
<td>dimmat</td>
<td>0 to force text inside, required by BricsCAD and AutoCAD</td>
</tr>
<tr>
<td>dimtad</td>
<td>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.</td>
</tr>
</tbody>
</table>

```python
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:

```python
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 \textit{angle}. For user defined locations it is not necessary to force text inside (\texttt{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')
```

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:

5.2. Tutorials 95
Center Mark/Lines

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

<table>
<thead>
<tr>
<th>dimcen</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>

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

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

See Linear Dimension Tutorial: *Overriding Measurement Text*

Measurement Text Formatting and Styling

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

### 5.2.21 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 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 possibles 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>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmove</td>
<td>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.</td>
</tr>
<tr>
<td>dimtix</td>
<td>1 to force text inside</td>
</tr>
<tr>
<td>dimmat-fit</td>
<td>0 to force text inside, required by BricsCAD and AutoCAD</td>
</tr>
<tr>
<td>dimtad</td>
<td>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.</td>
</tr>
</tbody>
</table>

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

![Diagram showing user defined text locations outside of the circle with different angle settings](image)
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})
```

![Diagram showing a circle with a dimension marked outside the circle and forced horizontal text.]

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')
```

![Diagram showing a circle with a dimension marked inside the circle.]

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})
```

![Diagram showing a circle with a dimension marked inside the circle and forced horizontal text.]

5.2. Tutorials
5.3 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 ezdxif, if you are looking for more information about the ezdxif internals look at the Reference section or if you want to learn how to use ezdxif go to the Tutorials section and for the solution of specific problems go to the Howto section.

5.3.1 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. ezdxif provides functions to create (new()) or modify (ezdxif.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)
5.3.2 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, uses 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

5.3.3 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)
```
| CONTINUOUS                                      |
| CENTER                                          |
| CENTERX2                                        |
| CENTER2                                         |
| DASHED                                          |
| DASHEDX2                                        |
| DASHED2                                         |
| PHANTOM                                         |
| PHANTOMX2                                       |
| PHANTOM2                                        |
| DASHDOT                                        |
| DASHDOTX2                                      |
| DASHDOT2                                       |
| DOT                                             |
| DOTX2                                          |
| DOT2                                           |
| DIVIDE                                         |
| DIVIDEX2                                        |
| DIVIDE2                                         |
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.

### 5.3.4 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)
cos = 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.

**5.3.5 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**

- DXF Reference for Arbitrary Axis Algorithm provided by Autodesk.

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 `Vector`):

```python
Az = Vector(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 = Vector(0, 1, 0).cross(Az).normalize()  # the cross-product operator
else:
    Ax = Vector(0, 0, 1).cross(Az).normalize()  # the cross-product operator
Ay = Az.cross(Ax).normalize()
```

**WCS to OCS**
```python
def wcs_to_ocs(point):
    px, py, pz = Vector(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 Vector(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))

def ocs_to_wcs(point):
    px, py, pz = Vector(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 Vector(x, y, z)
```

## 5.4 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.

### 5.4.1 DXF Structures

**Drawing**

The **Drawing** class manages all entities and tables related to a DXF drawing. You can read/write DXF drawings from/to file-system or from/to a text-stream.

**Drawing Management**

**Create New Drawings**

```python
ezdxf.new(dxfformat='AC1027', setup=None) \rightarrow Drawing
```

Create a new **Drawing** from scratch, `dxfformat` can be either 'AC1009' the official DXF version name or 'R12' the AutoCAD release name.

`new()` can create drawings for following DXF versions:
New in version 0.7.4: release name as DXF version

**Parameters**

- **dxfversion** – DXF version specifier as string, default is 'AC1027' (R2013)
- **setup** – setup drawing standard styles
  - None or False for no setup
  - 'all' or True to setup everything
  - a list of topics as strings, e.g. ['linetypes', 'styles'] to setup only linetypes and text styles:

| linetypes | setup line types |
| styles   | setup text styles |
| dimstyles | setup all dimension styles |
| dimstyles:metric | setup metric dimension styles |
| dimstyles:imperial | setup imperial dimension styles (not implemented yet) |
| visualstyles | setup 25 standard visual styles |

**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:

<table>
<thead>
<tr>
<th>Version</th>
<th>Release</th>
<th>Encoding</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; AC1009</td>
<td></td>
<td>$DWGCODEPAGE</td>
<td>pre AutoCAD R12 upgraded to AC1009</td>
</tr>
<tr>
<td>AC1009</td>
<td>R12</td>
<td>$DWGCODEPAGE</td>
<td>AutoCAD R12</td>
</tr>
<tr>
<td>AC1012</td>
<td>R13</td>
<td>$DWGCODEPAGE</td>
<td>AutoCAD R13 upgraded to AC1015</td>
</tr>
<tr>
<td>AC1014</td>
<td>R14</td>
<td>$DWGCODEPAGE</td>
<td>AutoCAD R14 upgraded to AC1015</td>
</tr>
<tr>
<td>AC1015</td>
<td>R2000</td>
<td>$DWGCODEPAGE</td>
<td>AutoCAD R2000</td>
</tr>
<tr>
<td>AC1018</td>
<td>R2004</td>
<td>$DWGCODEPAGE</td>
<td>AutoCAD R2004</td>
</tr>
<tr>
<td>AC1021</td>
<td>R2007</td>
<td>UTF-8</td>
<td>AutoCAD R2007</td>
</tr>
<tr>
<td>AC1024</td>
<td>R2010</td>
<td>UTF-8</td>
<td>AutoCAD R2010</td>
</tr>
<tr>
<td>AC1027</td>
<td>R2013</td>
<td>UTF-8</td>
<td>AutoCAD R2013</td>
</tr>
<tr>
<td>AC1032</td>
<td>R2018</td>
<td>UTF-8</td>
<td>AutoCAD R2018</td>
</tr>
</tbody>
</table>

**ezdxf.**readfile**(filename: str, encoding: str = None, legacy_mode: bool = False, filter_stack=None) → Drawing**

Read DXF document specified by filename from file-system.
This is the preferred method to open existing ASCII or Binary DXF files. Read the DXF drawing from the file-system with auto-detection of encoding for ASCII DXF files. 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 argument `legacy_mode` is True, `ezdxf` tries to reorder the coordinates of the LINE entity in files from CAD applications which wrote the coordinates in the order: x1, x2, y1, y2. Additional fixes may be added later. The legacy mode has a speed penalty of around 5%.

**Hint:** Try argument `legacy_mode=True` if error 'Missing required y coordinate near line: ...' occurs.

**Parameters**

- `filename` – filename of ASCII or Binary DXF document
- `encoding` – use None for auto detect (default), or set a specific encoding like 'utf-8', ignored for Binary DXF files
- `legacy_mode` – adds an extra trouble shooting import layer if True
- `filter_stack` – interface to put filters between reading layers

**Raises**

- IOError – File `filename` is not a DXF file or does not exist.
- DXFStructureError – for invalid DXF structure

```python
ezdxf.read(stream: TextIO, legacy_mode: bool = False, filter_stack=None) -> Drawing
```

Read DXF drawing from a text-stream. Open stream in text mode (`mode='rt'`) and the correct encoding has to be set at the open function, 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 required encoding.

If argument `legacy_mode` is True, `ezdxf` tries to reorder the coordinates of the LINE entity in files from CAD applications which wrote the coordinates in the order: x1, x2, y1, y2. Additional fixes may be added later. The legacy mode has a speed penalty of around 5%.

**Parameters**

- `stream` – input text stream opened with correct encoding, requires only a `readline()` method.
- `legacy_mode` – adds an extra trouble shooting import layer if True
- `filter_stack` – interface to put filters between reading layers

**Raises**

- DXFStructureError – for invalid DXF structure

```python
ezdxf.readzip(zipfile: str, filename: str = None) -> Drawing
```

Read DXF drawing specified by `filename` from a zip archive, or if `filename` is None the first DXF file in the zip archive.

**Parameters**

- `zipfile` – name of the zip archive
- `filename` – filename of DXF file, or None to read the first DXF file from the zip archive.

```python
ezdxf.decode_base64 (data: bytes) -> Drawing
```

Load DXF document from base64 encoded binary data, like uploaded data to web applications.
**Parameters** data – DXF document base64 encoded binary data

New in version 0.12: Thanks to Joseph Flack

**Save Drawings**

Save the drawing to the file system by *Drawing* methods `save()` or `saveas()`. Write the *Drawing* to a text stream with `write()`, the text stream requires at least a `write()` method.

New in version 0.11: Get required output encoding for text streams by *Drawing* property `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>$ACADVER</td>
<td>DXF version</td>
</tr>
<tr>
<td>$STDCREATE</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>$TDUPDATE</td>
<td>actual date/time at saving</td>
</tr>
<tr>
<td>$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**

*class ezdxf.drawing.Drawing*

The *Drawing* class manages all entities and tables related to a DXF drawing.

**dxfversion**

Actual DXF version like 'AC1009', set by *ezdxf.new()* or *ezdxf.readfile()*.

For supported DXF versions see *Drawing 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*

<table>
<thead>
<tr>
<th>supported</th>
<th>encodings</th>
</tr>
</thead>
<tbody>
<tr>
<td>'cp874'</td>
<td>Thai</td>
</tr>
<tr>
<td>'cp932'</td>
<td>Japanese</td>
</tr>
<tr>
<td>'gbk'</td>
<td>UnifiedChinese</td>
</tr>
<tr>
<td>'cp949'</td>
<td>Korean</td>
</tr>
<tr>
<td>'cp950'</td>
<td>TradChinese</td>
</tr>
<tr>
<td>'cp1250'</td>
<td>CentralEurope</td>
</tr>
<tr>
<td>'cp1251'</td>
<td>Cyrillic</td>
</tr>
<tr>
<td>'cp1252'</td>
<td>WesternEurope</td>
</tr>
<tr>
<td>'cp1253'</td>
<td>Greek</td>
</tr>
<tr>
<td>'cp1254'</td>
<td>Turkish</td>
</tr>
<tr>
<td>'cp1255'</td>
<td>Hebrew</td>
</tr>
<tr>
<td>'cp1256'</td>
<td>Arabic</td>
</tr>
<tr>
<td>'cp1257'</td>
<td>Baltic</td>
</tr>
<tr>
<td>'cp1258'</td>
<td>Vietnam</td>
</tr>
</tbody>
</table>

output_encoding
Returns required output encoding for saving to filesystem or encoding to binary data.

New in version 0.11.

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.

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: str, 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.
```

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.
```

New in version 0.12: Thanks to Joseph Flack

```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() -> Layout
    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.
```

```python
def 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>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm</td>
<td>Millimeter</td>
</tr>
<tr>
<td>cm</td>
<td>Centimeter</td>
</tr>
<tr>
<td>m</td>
<td>Meter (<em>ezdxf default</em>)</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[LayoutType]

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 actual '*Active' viewport configuration.

Parameters

- **height** – modelspace area to view
- **center** – modelspace location to view in the center of the CAD application window.

New in version 0.11.

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

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**

```python
class ezdxf.sections.header.HeaderSection
```

custom_vars
Stores the custom drawing properties in a `CustomVars` object.

```python
__len__() → int
```
Returns count of header variables.

```python
__contains__(key) → bool
```
Returns `True` if header variable `key` exist.

```python
varnames() → KeysView[KT]
```
Returns an iterable of all header variable names.

```python
get(key: str, default: Any = None) → Any
```
Returns value of header variable `key` if exist, else the `default` value.

```python
__getitem__(key: str) → Any
```
Get header variable `key` by index operator like: `drawing.header['$ACADVER']`

```python
__setitem__(key: str, value: Any) → None
```
Set header variable `key` to `value` by index operator like: `drawing.header['$ANGDIR'] = 1`

```python
__delitem__(key: str) → None
```
Delete header variable `key` by index operator like: `del drawing.header['$ANGDIR']`

```python
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.

```python
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)`.

```python
__len__() → int
```
Count of custom properties.

```python
__iter__() → Iterable[Tuple[str, str]]
```
Iterate over all custom properties as `(tag, value)` tuples.

```python
clear() → None
```
Remove all custom properties.
### 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

```python
class ezdxf.sections.classes.ClassesSection
```

```python
class ezdxf.entities.DXFClass
```

### ezdxf Documentation, Release 0.13

```python
def get(tag: str, default: str = None)
    Returns the value of the first custom property tag.

def has_tag(tag: str) -> bool
    Returns True if custom property tag exist.

def append(tag: str, value: str) -> None
    Add custom property as (tag, value) tuple.

def replace(tag: str, value: str) -> None
    Replaces the value of the first custom property tag by a new value. Raises DXFValueError if tag does not exist.

def remove(tag: str, all: bool = False) -> None
    Removes the first occurrence of custom property tag, removes all occurrences if all is True. Raises :class:`DXFValueError if tag does not exist.
```
**dxf.flags**

Proxy capabilities flag

<table>
<thead>
<tr>
<th>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>

**dxf.instance_count**

Instance count for a custom class.

**dxf.was_a_proxy**

Set to 1 if class was not loaded when this DXF file was created, and 0 otherwise.

**dxf.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*

```python
class ezdxfs.sections.tables.TablesSection
```

- **layers**
  
  `LayerTable` object for `Layer` objects

- **linetypes**
  
  Generic `Table` object for `Linetype` objects

- **styles**
  
  `StyleTable` object for `Textstyle` objects

- **dimstyles**
  
  Generic `Table` object for `DimStyle` objects

- **appids**
  
  Generic `Table` object for `AppID` 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]
Iterate 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.

<table>
<thead>
<tr>
<th>type_char</th>
<th>Anonymous Block Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>'U'</td>
<td>'U#####' anonymous BLOCK</td>
</tr>
<tr>
<td>'E'</td>
<td>'E#####' anonymous non-uniformly scaled BLOCK</td>
</tr>
<tr>
<td>'X'</td>
<td>'X#####' anonymous HATCH graphic</td>
</tr>
<tr>
<td>'D'</td>
<td>'D#####' anonymous DIMENSION graphic</td>
</tr>
<tr>
<td>'A'</td>
<td>'A#####' anonymous GROUP</td>
</tr>
<tr>
<td>'T'</td>
<td>'T#####' anonymous block for ACAD_TABLE content</td>
</tr>
</tbody>
</table>

rename_block(old_name: str, new_name: str) → None
Rename BlockLayout old_name to new_name

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

```python
def delete_all_blocks(safe: bool = True) -> None
    Delete all blocks except layout blocks (modelspace or paperspace). In safe mode, protected blocks are ignored silently.
```

Parameters **safe** – check if block is still referenced or special block without explicit references

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__(self) -> Iterable[DXFEntity]
        Iterable for all entities of modelspace and active paperspace.

    __len__(self) -> 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.
```
__len__() → int
Returns count of DXF objects.

__iter__() → Iterable[DXFObject]
Returns iterable of all DXF objects in the OBJECTS section.

__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].

__contains__(entity: Union[DXFObject, str]) → bool
Returns True if entity stored in OBJECTS section.

Parameters
- **entity** — DXFObject or handle as hex string

query(query: str = '*') → ezdxf.query.EntityQuery
Get all DXF objects matching the Entity Query String.

add_dictionary(owner: str = '0', hard_owned: bool = False) → ezdxf.entities.dictionary.Dictionary
Add new Dictionary object.

Parameters
- **owner** — handle to owner as hex string.
- **hard_owned** — True to treat entries as hard owned.

add_dictionary_with_default(owner='0', default='0', hard_owned: bool = False) → DictionaryWithDefault
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.

add_dictionary_var(owner: str = '0', value: str = '') → DictionaryVar
Add a new DictionaryVar object.

Parameters
- **owner** — handle to owner as hex string.
- **value** — value as string

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.

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.

<table>
<thead>
<tr>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm</td>
<td>Millimeter</td>
</tr>
<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>

5.4. Reference
(internal API), public interface `set_raster_variables()`

`set_wipeout_variables(frame: int = 0) → None`

Set wipeout variables.

Parameters `frame` – 0 = do not show image frame; 1 = show image frame

(Internal API), public interface `set_wipeout_variables()`

Tables

Table Classes

Generic Table Class

class `ezdxf.sections.table.Table`

Generic collection of table entries. Table entry names are case insensitive: 'Test' == 'TEST'.

  `static key(entity: Union[str, DXFEntity]) → str`

  Unified table entry key.

  `has_entry(name: Union[str, DXFEntity]) → bool`

  Returns True if an table entry name exist.

  `__contains__(name: Union[str, DXFEntity]) → bool`

  Returns True if an table entry name exist.

  `__len__() → int`

  Count of table entries.

  `__iter__() → Iterable[DXFEntity]`

  Iterable of all table entries.

  `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

  `get(name: str) → DXFEntity`

  Get table entry name (case insensitive). Raises DXFValueError if table entry does not exist.

  `remove(name: str) → None`

  Removes table entry name. Raises DXFValueError if table-entry does not exist.

  `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 ezdxf.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.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFEntity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'LAYER'</td>
</tr>
<tr>
<td>Factory function</td>
<td>Drawing.layers.new()</td>
</tr>
</tbody>
</table>

See also:
Layer Concept and Tutorial for Layers

class ezdxf.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)

dxf.flags

Layer flags (bit-coded values, feature for experts)
Layer is frozen; otherwise layer is thawed; use `is_frozen()`, `freeze()` and `thaw()`.

Layer is frozen by default in new viewports.

Layer is locked; use `is_locked()`, `lock()`, `unlock()`.

If set, table entry is externally dependent on an xref.

If both this bit and bit 16 are set, the externally dependent xref has been successfully resolved.

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.)

```
layer = Layer(name='layer1')
layer.is_frozen()  # True
layer.freeze()     # Layer is frozen
layer.thaw()       # Layer is thawed
```

dxf.color

Layer color, but use property `Layer.color` to get/set color value, because color is negative for layer status `off` (int).

dxf.true_color

Layer true color value as int, use property `Layer.rgb` to set/get true color value as `(r, g, b)` tuple.

(require DXF R2004)

dxf.linetype

Name of line type (str)

**Table:**

<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>

**Table:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>The table entry was referenced by at least one</td>
</tr>
<tr>
<td></td>
<td>entity in the drawing the last time the drawing</td>
</tr>
<tr>
<td></td>
<td>was edited. (This flag is for the benefit of</td>
</tr>
<tr>
<td></td>
<td>AutoCAD commands. It can be ignored by most</td>
</tr>
<tr>
<td></td>
<td>programs that read DXF files and need not be set</td>
</tr>
<tr>
<td></td>
<td>by programs that write DXF files)</td>
</tr>
</tbody>
</table>

```
layer = Layer(name='layer1')
layer.is_locked()  # False
layer.lock()       # Layer is locked
layer.unlock()     # Layer is thawed
```

dxf.lineweight

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.
```

(require DXF R13)

dxf.plotstyle_handle

Handle to plot style name?

(require DXF R13)

dxf.material_handle

Handle to default Material.

(require 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.

```
layers.rgb = (30, 40, 50)
rx, gy, bz = layers.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`.

New in version 0.10.
color
Get/set layer color, preferred method for getting the layer color, because *dxf.color* is negative for layer status *off*.
New in version 0.10.

description
Get/set layer description as string
New in version 0.10.

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.
New in version 0.10.

*is_frozen* () → bool
Returns True if layer is frozen.

freeze () → None
Freeze layer.

thaw () → None
Thaw layer.

*is_locked* () → bool
Returns True if layer is locked.

lock () → None
Lock layer, entities on this layer are not editable - just important in CAD applications.

unlock () → None
Unlock layer, entities on this layer are editable - just important in CAD applications.

*is_off* () → bool
Returns True if layer is off.

*is_on* () → bool
Returns True if layer is on.

on () → None
Switch layer on (visible).

off () → None
Switch layer off (invisible).

*get_color* () → int
Use property *Layer.color* instead.

*set_color* (*value: int*) → None
Use property *Layer.color* instead.

rename (*name: str*) → None
Rename layer and all known (documented) references to this layer.

**Warning:** Renaming layers may damage the DXF file in some circumstances!

**Parameters** *name* – new layer name

** Raises **
• ValueError – name contains invalid characters: <>":;?*|='
• ValueError – layer name already exist
• ValueError – renaming of layers '0' and 'DEFPOINTS' not possible

Style

Defines a text style (DXF Reference), can be used by entities: Text, Attrib and Attdef.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFEntity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'STYLE'</td>
</tr>
<tr>
<td>Factory function</td>
<td>Drawing.styles.new()</td>
</tr>
</tbody>
</table>

See also:
Tutorial for Text and DXF internals for DIMSTYLE Table.

class ezdxf.entities.Textstyle

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).

<table>
<thead>
<tr>
<th>1</th>
<th>If set, this entry describes a shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Vertical text</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)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>

dxf.height
    Fixed height in drawing units, 0 for not fixed (float).

dxf.width
    Width factor (float), default is 1.

dxf.oblique
    Oblique angle in degrees, 0 is vertical (float).

dxf.generation_flags
    Text generations flags (int)

<table>
<thead>
<tr>
<th>2</th>
<th>text is backward (mirrored in X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>text is upside down (mirrored in Y)</td>
</tr>
</tbody>
</table>
.. _linetype:

**Linetype**

Defines a linetype (DXF Reference).

.. csv-table::
   :header: Subclass of, ezdxf.entities.DXFEntity
   :header: DXF type, 'LTYPE'
   :header: Factory function, Drawing.linetypes.new()

See also:

.. _tutorial-linetype:

* Tutorial for Linetypes *

.. _dxf-internals-ltype:

* DXF Internals: LTYPE Table *

```
class ezdxf.entities.Linetype
```

.. _linetype-name:

**_name**

Linetype name (str).

.. _linetype-owner:

**_owner**

Handle to owner (Table).

.. _linetype-description:

**_description**

Linetype description (str).

.. _linetype-length:

**_length**

Total pattern length in drawing units (float).

.. _linetype-items:

**_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.

.. csv-table::
   :header: Subclass of, ezdxf.entities.DXFEntity
   :header: DXF type, 'DIMSTYLE'
   :header: Factory function, Drawing.dimstyles.new()

```
class ezdxf.entities.DimStyle
```
dxfs.owner
Handle to owner (Table).

dxfs.name
Dimension style name.

dxfs.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, 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>

dxfs.dimpost
Prefix/suffix for primary units dimension values.

dxfs.dimapost
Prefix/suffix for alternate units dimensions.

dxfs.dimblk
Block type to use for both arrowheads as name string.

dxfs.dimblk1
Block type to use for first arrowhead as name string.

dxfs.dimblk2
Block type to use for second arrowhead as name string.

dxfs.dimscale
Global dimension feature scale factor. (default=1)

dxfs.dimasz
Dimension line and arrowhead size. (default=0.25)

dxfs.dimexo
Distance from origin points to extension lines. (default imperial=0.0625, default metric=0.625)

dxfs.dimdli
Incremental spacing between baseline dimensions. (default imperial=0.38, default metric=3.75)

dxfs.dimexe
Extension line distance beyond dimension line. (default imperial=0.28, default metric=2.25)

dxfs.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.

dxfs.dimdle
Dimension line extension beyond extension lines. (default=0)

dxfs.dimtp
Upper tolerance value for tolerance dimensions. (default=0)

dxfs.dimtm
Lower tolerance value for tolerance dimensions. (default=0)

dxfs.dimtxt
Size of dimension text. (default imperial=0.28, default metric=2.5)
**ezdxf Documentation, Release 0.13**

- **dxf.dimcen**
  Controls placement of center marks or centerlines. (default imperial=0.09, default metric=2.5)

- **dxf.dimtsz**
  Controls size of dimension line tick marks drawn instead of arrowheads. (default=0)

- **dxf.dimlaf**
  Alternate units dimension scale factor. (default=25.4)

- **dxf.dimlfac**
  Scale factor for linear dimension values. (default=1)

- **dxf.dimtvp**
  Vertical position of text above or below dimension line if `dimtad` is 0. (default=0)

- **dxf.dimtfacl**
  Scale factor for fractional or tolerance text size. (default=1)

- **dxf.dimgap**
  Gap size between dimension line and dimension text. (default imperial=0.09, default metric=0.625)

- **dxf.dimaltrnd**
  Rounding value for alternate dimension units. (default=0)

- **dxf.dimaltol**
  Toggles creation of appended tolerance dimensions. (default imperial=1, default metric=0)

- **dxf.dimgap**
  Toggles creation of limits-style dimension text. (default=0)

- **dxf.dimtih**
  Orientation of text inside extension lines. (default imperial=1, default metric=0)

- **dxf.dimtoh**
  Orientation of text outside extension lines. (default imperial=1, default metric=0)

- **dxf.dimsel**
  Toggles suppression of first extension line. (default=0)

- **dxf.dimsel2**
  Toggles suppression of second extension line. (default=0)

- **dxf.dimtad**
  Sets vertical text placement relative to dimension line. (default imperial=0, default metric=1)

<table>
<thead>
<tr>
<th>Value</th>
<th>Placement</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>

- **dxf.dimzine**
  Zero suppression for primary units dimensions. (default imperial=0, default metric=8)

  Values 0-3 affect feet-and-inch dimensions only.
<table>
<thead>
<tr>
<th>Value</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>Value</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**. **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**. **dimdec**

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)
**dxfl.** `dimalttd`  
Decimal places for alternate units tolerance values. (default imperial=4, default metric=2)

**dxfl.** `dimaunit`  
Unit format for angular dimension values. (default=0)

**dxfl.** `dimfrac`  
Controls the fraction format used for architectural and fractional dimensions. (default=0)

**dxfl.** `dimlunit`  
Specifies units for all nonangular dimensions. (default=2)

**dxfl.** `dimsep`  
Specifies a single character to use as a decimal separator. (default imperial = ".", default metric = ",")  
This is an integer value, use `ord()` to write value.

**dxfl.** `dimtmove`  
Controls the format of dimension text when it is moved. (default=0)

<table>
<thead>
<tr>
<th>0</th>
<th>Moves the dimension line with dimension text</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Adds a leader when dimension text is moved</td>
</tr>
<tr>
<td>2</td>
<td>Allows text to be moved freely without a leader</td>
</tr>
</tbody>
</table>

**dxfl.** `dimjust`  
Horizontal justification of dimension text. (default=0)

<table>
<thead>
<tr>
<th>0</th>
<th>Center of dimension line</th>
</tr>
</thead>
<tbody>
<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>

**dxfl.** `dimsd1`  
Toggles suppression of first dimension line. (default=0)

**dxfl.** `dimsd2`  
Toggles suppression of second dimension line. (default=0)

**dxfl.** `dimtolj`  
Vertical justification for dimension tolerance text. (default=1)

<table>
<thead>
<tr>
<th>0</th>
<th>Align with bottom line of dimension text</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

**dxfl.** `dimtzen`  
Zero suppression for tolerances values, see `DimStyle.dxf.dimtzen`

**dxfl.** `dimaltz`  
Zero suppression for alternate units dimension values. (default=0)

**dxfl.** `dimalttz`  
Zero suppression for alternate units tolerance values. (default=0)

**dxfl.** `dimfit`  
Obsolete, now use `DIMATFIT` and `DIMTMOVE`
ezdxf Documentation, Release 0.13

```plaintext
dxf.\texttt{dimupt}
   Controls user placement of dimension line and text. (default=0)

dxf.\texttt{dimatfit}
   Controls placement of text and arrowheads when there is insufficient space between the extension lines.
   (default=3)

dxf.\texttt{dimtxtsty}
   Text style used for dimension text by name.

dxf.\texttt{dimtxtsty\_handle}
   Text style used for dimension text by handle of STYLE entry. (use DimStyle.dxf.dimtxtsty to
   get/set text style by name)

dxf.\texttt{dimldrblk}
   Specify arrowhead used for leaders by name.

dxf.\texttt{dimldrblk\_handle}
   Specify arrowhead used for leaders by handle of referenced block. (use DimStyle.dxf.dimldrblk
   to get/set arrowhead by name)

dxf.\texttt{dimblk\_handle}
   Block type to use for both arrowheads, handle of referenced block. (use DimStyle.dxf.dimblk
   to get/set arrowheads by name)

dxf.\texttt{dimblk1\_handle}
   Block type to use for first arrowhead, handle of referenced block. (use DimStyle.dxfdimblk1
   to get/set arrowhead by name)

dxf.\texttt{dimblk2\_handle}
   Block type to use for second arrowhead, handle of referenced block. (use DimStyle.dxf.dimblk2
   to get/set arrowhead by name)

dxf.\texttt{dimlwd}
   Lineweight value for dimension lines. (default=-2, BYBLOCK)

dxf.\texttt{dimlwe}
   Lineweight value for extension lines. (default=-2, BYBLOCK)

dxf.\texttt{dimltype}
   Specifies the linetype used for the dimension line as linetype name, requires DXF R2007+

dxf.\texttt{dimltype\_handle}
   Specifies the linetype used for the dimension line as handle to LTYPE entry, requires DXF R2007+
   (use DimStyle.dxf.dimltype to get/set linetype by name)

dxf.\texttt{dimltex1}
   Specifies the linetype used for the extension line 1 as linetype name, requires DXF R2007+

dxf.\texttt{dimltex1\_handle}
   Specifies the linetype used for the extension line 1 as handle to LTYPE entry, requires DXF R2007+
   (use DimStyle.dxf.dimltex1 to get/set linetype by name)

dxf.\texttt{dimltex2}
   Specifies the linetype used for the extension line 2 as linetype name, requires DXF R2007+

dxf.\texttt{dimlex2\_handle}
   Specifies the linetype used for the extension line 2 as handle to LTYPE entry, requires DXF R2007+
   (use DimStyle.dxf.dimlex2 to get/set linetype by name)

dxf.\texttt{dimfxlon}
   Extension line has fixed length if set to 1, requires DXF R2007+
```

5.4. Reference
dx.f.*dimfxl*
Length of extension line below dimension line if fixed (DimStyle.dxf.dimtfxlon == 1), DimStyle.dxf.dimexen defines the length above the dimension line, requires DXF R2007+

dx.f.*dimtfill*
Text fill 0=off; 1=background color; 2=custom color (see DimStyle.dxf.dimtfillclr), requires DXF R2007+

dx.f.*dimtfillclr*
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 dwg.

*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_text_format* (prefix: str = ", postfix: str = ", rnd: float = None, dec: int = None, sep: str = None, leading_zeros: bool = True, trailing_zeros: bool = True) → 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 (requires DXF R2000)
- **leading_zeros** – suppress leading zeros for decimal dimensions if False
- **trailing_zeros** – suppress trailing zeros for decimal dimensions if False
**set_dimline_format**

Set dimension line properties

- **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**

Set common extension line attributes.

- **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**

Set extension line 1 attributes.

- **linetype** – linetype for extension line 1 (requires DXF R2007)
- **disable** – disable extension line 1 if True

**set_extline2**

Set extension line 2 attributes.

- **linetype** – linetype for extension line 2 (requires DXF R2007)
- **disable** – disable extension line 2 if True

**set_tolerance**

Set tolerance text format, upper and lower value, text height factor, number of decimal places or leading and trailing zero suppression.

- **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)

```py
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) 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>
<th>DXF type</th>
<th>'VPORT'</th>
<th>Factory function</th>
<th><code>Drawing.viewports.new()</code></th>
</tr>
</thead>
</table>

See also:

DXF Internals: *VPORT Configuration Table*

```py
class ezdxf.entities.VPort
```

Subclass of `DXFEntity`

Defines a viewport configurations for the modelspace.

- **dxf.owner**
  - Handle to owner (`ViewportTable`).

- **dxf.name**
  - Viewport name

- **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, 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.lower_left
   Lower-left corner of viewport

dxf.upper_right
   Upper-right corner of viewport

dxf.center
   View center point (in DCS)

dxf.snap_base
   Snap base point (in DCS)

dxf.snap_spacing
   Snap spacing X and Y

dxf.grid_spacing
   Grid spacing X and Y

dxf.direction_point
   View direction from target point (in WCS)

dxf.target_point
   View target point (in WCS)

dxf.height
   View height

dxf.aspect_ratio

dxf.lens_length
   Lens focal length in mm

dxf.front_clipping
   Front clipping plane (offset from target point)

dxf.back_clipping
   Back clipping plane (offset from target point)

dxf.snap_rotation
   Snap rotation angle in degrees

dxf.view_twist
   View twist angle in degrees

dxf.status

dxf.view_mode

dxf.circle_zoom

dxf.fast_zoom

dxf.ucs_icon

dxf.snap_on

dxf.grid_on

dxf.snap_style

dxf.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><code>ezdxf.entities.DXFEntity</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'VIEW'</td>
</tr>
<tr>
<td>Factory function</td>
<td><code>Drawing.views.new()</code></td>
</tr>
</tbody>
</table>

See also:

DXF Internals: VIEW Table

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></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></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></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>Bit</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>

UCS

Defines a 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>

**dxf.origin**

Origin as \((x, y, z)\) tuple

**dxf.xaxis**

X-axis direction as \((x, y, z)\) tuple

**dxf.yaxis**

Y-axis direction as \((x, y, z)\) tuple

**ucs()** → UCS

Returns an `ezdxf.math.ucs` object for this UCS table entry.

**BlockRecord**

BLOCK_RECORD (DXF Reference) is a basic management structure for `BlockLayout` and `Layout`. `ezdxf` take care of correct BLOCK_RECORDS management, users don’t have to worry about.

<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

---

5.4. Reference
<table>
<thead>
<tr>
<th>0</th>
<th>Unitless</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Inches</td>
</tr>
<tr>
<td>2</td>
<td>Feet</td>
</tr>
<tr>
<td>3</td>
<td>Miles</td>
</tr>
<tr>
<td>4</td>
<td>Millimeters</td>
</tr>
<tr>
<td>5</td>
<td>Centimeters</td>
</tr>
<tr>
<td>6</td>
<td>Meters</td>
</tr>
<tr>
<td>7</td>
<td>Kilometers</td>
</tr>
<tr>
<td>8</td>
<td>Microinches</td>
</tr>
<tr>
<td>9</td>
<td>Mils</td>
</tr>
<tr>
<td>10</td>
<td>Yards</td>
</tr>
<tr>
<td>11</td>
<td>Angstroms</td>
</tr>
<tr>
<td>12</td>
<td>Nanometers</td>
</tr>
<tr>
<td>13</td>
<td>Microns</td>
</tr>
<tr>
<td>14</td>
<td>Decimeters</td>
</tr>
<tr>
<td>15</td>
<td>Decameters</td>
</tr>
<tr>
<td>16</td>
<td>Hectometers</td>
</tr>
<tr>
<td>17</td>
<td>Gigameters</td>
</tr>
<tr>
<td>18</td>
<td>Astronomical units</td>
</tr>
<tr>
<td>19</td>
<td>Light years</td>
</tr>
<tr>
<td>20</td>
<td>Parsecs</td>
</tr>
<tr>
<td>21</td>
<td>US Survey Feet</td>
</tr>
<tr>
<td>22</td>
<td>US Survey Inch</td>
</tr>
<tr>
<td>23</td>
<td>US Survey Yard</td>
</tr>
<tr>
<td>24</td>
<td>US Survey Mile</td>
</tr>
</tbody>
</table>

**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 <em>BlockLayout</em>)</td>
</tr>
</tbody>
</table>

See also:

*Tutorial for Blocks* and DXF Internals: *Block Management Structures*

```python
class ezdxf.entities.Block
```
**dxfl.handle**
- BLOCK handle as plain hex string. (feature for experts)

**dxfl.owner**
- Handle to owner as plain hex string. (feature for experts)

**dxfl.layer**
- Layer name as string; default value is '0'

**dxfl.name**
- BLOCK name as string. (case insensitive)

**dxfl.name2**
- The same `dxfl.name` a second time (meaning?)

**dxfl.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.

**dxfl.flags**
- BLOCK flags (bit-coded)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Anonymous block generated by hatching, associative dimensioning, other internal operations, or an application</td>
</tr>
<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>

**dxfl.xref_path**
- File system path as string, if this block defines an external reference (XREF).

**is_layout_block**
- Returns True if this is a *Modelspace* or *Paperspace* block definition.

**is_anonymous**
- Returns True if this is an anonymous block generated by hatching, associative dimensioning, other internal operations, or an application.

  New in version 0.12.

**is_xref**
- Returns True if bock is an external referenced file. (XREF)

  New in version 0.12.

**is_xref_overlay**
- Returns True if bock is an external referenced overlay file. (XREF)

  New in version 0.12.
**EndBlk**

ENDBLK entity is embedded into the `BlockLayout` object. The ENDBLK entity is accessible by the `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>

```python
class ezdxf.entities.EndBlk
```

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; should always be the same as `Block.dxf.layer`

**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><code>ezdxf.layouts.BaseLayout.add_blockref()</code></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!

TODO: influence of layer, linetype, color DXF attributes to block entities

```python
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 (int)

**dxf.row_spacing**
Distance between two insert points in row direction (float)

**dxf.column_count**
Count of repeated insertions in column direction (int)

**dxf.column_spacing**
Distance between two insert points in column direction (float)

**attrs**
A list of all attached *Attrib* entities.

**has_scaling**
Returns True if any axis scaling is applied.

New in version 0.12.

**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.

New in version 0.12.

**set_scale** (*factor: float*)
Set uniform scaling.

New in version 0.12.

**block** () → Optional[BlockLayout]
Returns associated *BlockLayout*.

New in version 0.12.

**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
**ezdxf Documentation, Release 0.13**

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

Returns ATTRIB or ATTDEF object

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.

New in version 0.11.

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.

New in version 0.13.

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, returns self (floating interface).

New in version 0.13.

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:

```python
doc.entitydb.add(entity)
msp = doc.modelspace()
msp.add_entity(entity)
```

**Warning:** Non uniform scaling returns incorrect results for text entities (TEXT, MTEXT, ATTRIB) and some other entities like HATCH with arc or ellipse path segments.

Parameters skipped_entity_callback – called whenever the transformation of an entity is not supported and so was skipped

New in version 0.12: experimental feature

Changed in version 0.13: deprecated non_uniform_scaling argument
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.

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. Multiple inserts by row and column attributes is not supported.

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 lead to incorrect results for text entities (TEXT, MTEXT, ATTRIB) and some other entities like HATCH with arc or ellipse path segments.

**Parameters** target_layout – target layout for exploded entities, None for same layout as source entity.

New in version 0.12: experimental feature

Changed in version 0.13: deprecated non_uniform_scaling argument

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>ezdxf.entities.Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'ATTRIB'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_attrib() (stand alone entity)</td>
</tr>
<tr>
<td>Factory function</td>
<td>Insert.add_attrib() (attached to Insert)</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.Attrib

ATTRIB 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)

is_invisible

Attribute is invisible (does not appear).
```python
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.

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() → Iterable[str]

Returns iterable of all layout names.

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`.

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
- `new_name` – new layout name

Raises

- `DXFValueError` – try to rename 'Model'
- `DXFValueError` – Layout `new_name` already exist.

delete(name: str) → None

Delete layout `name` and all entities owned by it.

Parameters

- `name` (str) – layout name as shown in tabs
Raises

- KeyError – if layout name do not exists
- ValueError – if name is 'Model', deleting modelspace 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

```python
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.

__len__() \rightarrow int
   Returns count of entities owned by the layout.

__iter__() \rightarrow Iterable[ezdxf.entities.dxfgfx.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() \rightarrow Dictionary
   Returns the associated extension dictionary, creates a new one if necessary.

delete_entity(entity: DXFGraphic) \rightarrow None
   Delete entity from layout entity space and the entity database, this destroys the entity.

delete_all_entities() \rightarrow None
   Delete all entities from layout entity space and from entity database, this destroys all entities in this layout.

unlink_entity(entity: DXFGraphic) \rightarrow 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 = '*') \rightarrow EntityQuery
   Get all DXF entities matching the Entity Query String.

groupby(dxffattrib: str = '', key: KeyFunc = None) \rightarrow dict
   Returns a dict of entity lists, where entities are grouped by a dxffattrib or a key function.

   Parameters
   - dxffattrib – 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) \rightarrow 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) \rightarrow 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) \rightarrow 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

New in version 0.11.1.

add_point (location: Vertex, dxfattribs: dict = 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 = 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

```python
add_circle(center: Vertex, radius: float, dxfattribs: dict = 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

```python
add_ellipse(center: Vertex, major_axis: Vertex = (1, 0, 0), ratio: float = 1, start_param: float = 0, end_param: float = 6.283185307179586, dxfattribs: dict = 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
- **start_param** – start of ellipse curve
- **end_param** – end param of ellipse curve
- **dxfattribs** – additional DXF attributes

### add_arc

```python
add_arc(center: Vertex, radius: float, start_angle: float, end_angle: float, is_counter_clockwise: bool = True, dxfattribs: dict = 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

```python
add_solid(points: Iterable[Vertex], dxfattribs: dict = 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

```python
add_trace(points: Iterable[Vertex], dxfattribs: dict = 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
• **points** – iterable of 3 or 4 2D/3D points in WCS
  • **dxfattribs** (*dict*) – additional DXF attributes for Trace entity

**add_3dface** (*points*: *Iterable*[Vertex], *dxfattribs*: *dict* = 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* = None) → Text
Add a Text entity, see also **Style**.

**Parameters**
  • **text** – content string
  • **dxfattribs** (*dict*) – additional DXF attributes for Text entity

**add_blockref** (*name*: str, *insert*: Vertex, *dxfattribs*: *dict* = None) → Insert
Add an **Insert** entity.

**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* = 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**
  • **name** – block name
  • **insert** – insert location as 2D/3D point in WCS
  • **values** – **Attrib** tag values as **tag/value** pairs
  • **dxfattribs** – additional DXF attributes for Insert entity

**add_attrib** (*tag*: str, *text*: str, *insert*: Vertex = (0, 0), *dxfattribs*: *dict* = None) → Attrib
Add an **Attrib** as stand alone DXF entity.

**Parameters**
  • **tag** – tag name as string
  • **text** – tag value as string
  • **insert** – insert location as 2D/3D point in WCS
  • **dxfattribs** – additional DXF attributes for Attrib entity
add_attdef (tag: str, insert: Vertex = (0, 0), text: str = '', dxfattribs: dict = None) → AttDef
Add an AttDef as stand alone DXF entity.
Set position and alignment by the idiom:

```
layout.add_attdef('NAME').set_pos((2, 3), align='MIDDLE_CENTER')
```

Parameters

- **tag** – tag name as string
- **insert** – insert location as 2D/3D point in WCS
- **text** – tag value as string
- **dxfattribs** – additional DXF attributes

add_polyline2d (points: Iterable[Vertex], dxfattribs: dict = None, format: str = None) → Polyline
Add a 2D Polyline entity.

Parameters

- **points** – iterable of 2D points in WCS
- **dxfattribs** – additional DXF attributes
- **format** – user defined point format like add_lwpolyline(), default is None

New in version 0.11: user defined point format

add_polyline3d (points: Iterable[Vertex], dxfattribs: dict = None) → Polyline
Add a 3D Polyline entity.

Parameters

- **points** – iterable of 3D points in WCS
- **dxfattribs** – additional DXF attributes

add_polymesh (size: Tuple[int, int] = (3, 3), dxfattribs: dict = None) → Polymesh
Add a Polymesh entity, which is a wrapper class for the POLYLINE entity. A polymesh is a grid of \( m \times n \) vertices and every vertex has its own \((x, y, z)\)-coordinates.

Parameters

- **size** – 2-tuple \((m, n)\)
- **dxfattribs** – additional DXF attributes for Polyline entity

add_polyface (dxfattribs: dict = None) → 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 = None) → 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 = 'xyseb', dxfattribs: dict = None) → LW-Polyline
Add a 2D polyline as LWPolyline entity. A points are defined as (x, y, [start_width, [end_width, [bulge]]) tuples, but order can be redefined by the format argument. Set start_width, end_width to 0 to be ignored like (x, y, 0, 0, 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-axis is ignored)

Parameters
• points – iterable of (x, y, [start_width, [end_width, [bulge]]) tuples
• format – user defined point format, default is "xyseb"
• dxfattribs – additional DXF attributes

add_mtext(text: str, dxfattribs: dict = None) → 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 = 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 = 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_spline(fit_points: Iterable[Vertex] = None, degree: int = 3, dxfattribs: dict = None) → Spline
Add a B-spline (Spline entity) defined by 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. (requires DXF R2000)

AutoCAD creates a spline through fit points by a proprietary algorithm. `ezdxf` can not reproduce the control point calculation. See also: Tutorial for Spline.

**Parameters**

- `fit_points` – iterable of fit points as `(x, y[, z])` in WCS, create ‘empty’ Spline if `None`
- `degree` – degree of B-spline
- `dxfattribs` – additional DXF attributes

**add_spline_control_frame** *(fit_points: Iterable[Vertex], degree: int = 3, method: str = ‘chord’, dxfattribs: dict = None) → Spline*

Add a Spline entity passing through given fit points by global B-spline interpolation, the new SPLINE entity is defined by a control frame and not by the fit points.

- “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.

**Parameters**

- `fit_points` – iterable of fit points as `(x, y[, z])` in WCS
- `degree` – degree of B-spline
- `method` – calculation method for parameter vector t
- `dxfattribs` – additional DXF attributes

**add_open_spline** *(control_points: Iterable[Vertex], degree: int = 3, knots: Iterable[float] = None, dxfattribs: dict = 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
- `knots` – knot values as iterable of floats
- `dxfattribs` – additional DXF attributes

**add_closed_spline** *(control_points: Iterable[Vertex], degree: int = 3, knots: Iterable[float] = None, dxfattribs: dict = None) → Spline*

Add a closed uniform Spline defined by control_points. (requires DXF R2000)

Closed uniform B-splines is a closed curve start and end at the first control point.

**Parameters**

- `control_points` – iterable of 3D points in WCS
- `degree` – degree of B-spline
• **knots** – knot values as iterable of floats
• **dxfattribs** – additional DXF attributes

```python
add_rational_spline(control_points: Iterable[Vertex], weights: Sequence[float], degree: int = 3, knots: Iterable[float] = None, dxfattribs: dict = 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
• **knots** – knot values as iterable of floats
• **dxfattribs** – additional DXF attributes

```python
add_closed_rational_spline(control_points: Iterable[Vertex], weights: Sequence[float], degree: int = 3, knots: Iterable[float] = None, dxfattribs: dict = None) → Spline
```

Add a closed 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.

Closed rational uniform B-splines start and end at the first control point.

**Parameters**

• **control_points** – iterable of 3D points in WCS
• **weights** – weight values as iterable of floats
• **degree** – degree of B-spline
• **knots** – knot values as iterable of floats
• **dxfattribs** – additional DXF attributes

```python
add_hatch(color: int = 7, dxfattribs: dict = None) → 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 = None) → 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 = None) → 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

**add_underlay** *(underlay_def: UnderlayDef, insert: Vertex = (0, 0, 0), scale=(1, 1, 1), rotation: float = 0.0, dxfattribs: dict = None) → Underlay*

Add a *Underlay* entity, requires a *UnderlayDef* entity, see *Tutorial for Underlay and UnderlayDefinition*. (requires DXF R2000)

**Parameters**

• **underlay_def** – required underlay definition as *UnderlayDef*
• **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_linear_dim** *(base: Vertex, p1: Vertex, p2: Vertex, location: Vertex = None, text: str = '<>', angle: float = 0, text_rotation: float = None, dimstyle: str = 'EZDXF', override: dict = None, dxfattribs: dict = None) → DimStyleOverride*

Add a 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.
• **dxfattribs** – additional DXF attributes for *Dimension* entity

Returns: *DimStyleOverride*

```python
add_multi_point_linear_dim(base: Vertex, points: Iterable[Vertex], angle: float = 0, ucs: UCS = None, avoid_double_rendering: bool = True, dimstyle: str = 'EZDXF', override: dict = None, dxfattribs: dict = None, discard=False) → None
```

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 = None, dxfattribs: dict = 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**
• \textbf{p1} – measurement point 1 and start point of extension line 1 (in UCS)
• \textbf{p2} – measurement point 2 and start point of extension line 2 (in UCS)
• \textbf{distance} – distance of dimension line from measurement points
• \textbf{text} – None or “<>” the measurement is drawn as text, “ ” (one space) suppresses the dimension text, everything else \textit{text} is drawn as dimension text
• \textbf{dimstyle} – dimension style name (\textit{DimStyle} table entry), default is 'EZDXF'
• \textbf{override} – DimStyleOverride attributes
• \textbf{dxfattribs} – DXF attributes for Dimension entity

Returns: DimStyleOverride

\textbf{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 = None, dxfattribs: dict = None) \rightarrow \text{DimStyleOverride}

Add a radius Dimension line. The radius dimension line requires a \textit{center} point and a point \textit{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 DimStyleOverride object - to create the necessary dimension geometry, you have to call \textit{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: 'EZ_RADIUS'
• Default text location outside horizontal: 'EZ_RADIUS' + dimtoh=1
• Default text location inside: text aligned with dimension line; dimension style: 'EZ_RADIUS_INSIDE'
• Default text location inside horizontal: 'EZ_RADIUS_INSIDE' + dimtih=1
• User defined text location: argument \textit{location} \(!=\) None, text aligned with dimension line; dimension style: 'EZ_RADIUS'
• User defined text location horizontal: argument \textit{location} \(!=\) None, 'EZ_RADIUS' + dimtoh=1 for text outside horizontal, '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:} ezdxf ignores some DIMSTYLE variables, so render results may differ from BricsCAD or AutoCAD.

\textbf{Parameters}

• \textbf{center} – center point of the circle (in UCS)
• **mpoint** – measurement point on the circle, overrides `angle` and `radius` (in UCS)
• **radius** – radius in drawing units, requires argument `angle`
• **angle** – specify angle of dimension line in degrees, requires argument `radius`
• **location** – user defined dimension text location, overrides `mpoint` and `angle`, but requires `radius` (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_2p** (center: Vertex, mpoint: Vertex, text: str = '<>', dimstyle: str = 'EZ_RADIUS', override: dict = None, dxfattribs: dict = 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


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 = None, dxfattribs: dict = 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 'EZDXF'

• **override** – DimStyleOverride attributes

• **dxfattribs** – additional DXF attributes for Dimension entity

**Returns:** DimStyleOverride

(not implemented yet!)

**add_diameter_dim_2p**(p1: Vertex, p2: Vertex, text: str = '<>', dimstyle: str = 'EZ_RADIUS', override: dict = None, dxfattribs: dict = None) → DimStyleOverride

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 (vertices: Iterable[Vertex], dimstyle: str = 'EZDXF', override: dict = None, dxfattribs: dict = None) → 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

• vertices – leader vertices (in WCS)
• dimstyle – dimension style name (DimStyle table entry), default is 'EZDXF'
• override – override DimStyleOverride attributes
• dxfattribs – additional DXF attributes for Leader entity

add_body (acis_data: str = None, dxfattribs: dict = None) → Body

Add a Body entity. (requires DXF R2000)

Parameters

• acis_data – ACIS data as iterable of text lines as strings, no interpretation by ezdxf
  possible
• dxfattribs – additional DXF attributes

add_region (acis_data: str = None, dxfattribs: dict = None) → Region

Add a Region entity. (requires DXF R2000)

Parameters

• acis_data – ACIS data as iterable of text lines as strings, no interpretation by ezdxf
  possible
• dxfattribs – additional DXF attributes

add_3dsolid (acis_data: str = None, dxfattribs: dict = None) → Solid3d

Add a 3DSOLID entity (Solid3d). (requires DXF R2000)

Parameters
• **acis_data** – ACIS data as iterable of text lines as strings, no interpretation by ezdxf possible

• **dxfattribs** – additional DXF attributes

**add_surface** *(acis_data: str = None, dxfattribs: dict = None) → Surface*

Add a *Surface* 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_extruded_surface** *(acis_data: str = None, dxfattribs: dict = None) → 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 = 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 = 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 = 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.
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__ (entity: Union[DXFGraphic, str]) → bool

Returns True if entity is stored in this layout.

Parameters entity – DXFGraphic object or handle as hex string

reset_extends () → None

Reset extends.

set_plot_type (value: int = 5) → None

| 0 | last screen display |
| 1 | drawing extents     |
| 2 | drawing limits      |
| 3 | view specific (defined by Layout.dxf.plot_view_name) |
| 4 | window specific (defined by Layout.set_plot_window_limit()) |
| 5 | layout information (default) |

Parameters value – plot type

Raises DXFValueError – for value out of range

set_plot_style (name: str = 'ezdxf.ctb', show: bool = False) → None

Set plot style file of type .ctb.

Parameters

• name – plot style filename
• show – show plot style effect in preview? (AutoCAD specific attribute)

set_plot_window (lower_left: Tuple[float, float] = (0, 0), upper_right: Tuple[float, float] = (0, 0)) → None

Set plot window size in (scaled) paper space units.

Parameters

• lower_left – lower left corner as 2D point
• upper_right – upper right corner as 2D point

set_redraw_order (handles: Union[Dict[KT, VT], Iterable[Tuple[str, str]]]) → None

If the header variable $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. handles can be a dict of object_handle and sort_handle as (key, value) pairs, or an iterable of (object_handle, sort_handle) tuples.

The sort_handle doesn’t have to be unique, same or all handles can share the same sort_handle and sort_handles can use existing handles too.

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; for iterable an association is a tuple (object_handle, sort_handle); for dict the association is key: object_handle, value: sort_handle
get_redraw_order() \rightarrow \text{Iterable}[\text{Tuple}[\text{str}, \text{str}]]

Returns iterable for all existing table entries as (object_handle, sort_handle) pairs. (see also
set_redraw_order())

plot_viewportBorders (state: bool = True) \rightarrow \text{None}

show_plot_styles (state: bool = True) \rightarrow \text{None}

plot_centered (state: bool = True) \rightarrow \text{None}

plot_hidden (state: bool = True) \rightarrow \text{None}

use_standard_scale (state: bool = True) \rightarrow \text{None}

use_plot_styles (state: bool = True) \rightarrow \text{None}

scale_lineweights (state: bool = True) \rightarrow \text{None}

print_lineweights (state: bool = True) \rightarrow \text{None}

draw_viewports_first (state: bool = True) \rightarrow \text{None}

model_type (state: bool = True) \rightarrow \text{None}

update_paper (state: bool = True) \rightarrow \text{None}

zoom_to_paper_on_update (state: bool = True) \rightarrow \text{None}

plot_flags_initializing (state: bool = True) \rightarrow \text{None}

prev_plot_init (state: bool = True) \rightarrow \text{None}

set_plot_flags (flag, state: bool = True) \rightarrow \text{None}

### Modelspace

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) \rightarrow \text{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 () \rightarrow \text{Optional}[\text{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, extends 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** – int 0-32 = standard scale type or tuple(numerator, denominator) e.g. (1, 50) for 1:50
- **name** – paper name prefix '{name}_({width}_x_{height}_{unit})'
- **device** – device .pc3 configuration file or system printer name

**rename**
```
(name: str) → None
```
Rename layout to **name**, changes the name displayed in tabs by CAD applications, not the internal BLOCK name.

**viewports**
```
() → List[DXFGraphic]
```
Get all VIEWPORT entities defined in paperspace layout. Returns a list of **Viewport** objects, sorted by id, the first entity is always the paperspace view with an id of 1.

**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 add a new main viewport.

**reset_paper_limits**
```
() → None
```
Set paper limits to default values, all values in paperspace units but without plot scale (?).

**get_paper_limits**
```
() → Tuple[Tuple[float, float], Tuple[float, float]]
```
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 ((x1, y1), (x2, y2)), lower left corner is (x1, y1), upper right corner is (x2, y2).
### BlockLayout

**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]*) → bool
  - Returns *True* if block contains *entity*.

        Parameters
        entity – DXFGraphic object or handle as hex string

- **attdefs** () → Iterable[AttDef]
  - Returns iterable of all *Attdef* entities.

- **has_attdef** (*tag: str*) → bool
  - Returns *True* if an *Attdef* for *tag* exist.

- **get_attdef** (*tag: str*) → Optional[DXFGraphic]
  - Returns attached *Attdef* entity by *tag* name.

- **get_attdef_text** (*tag: str, default: str = None*) → str
  - Returns text content for *Attdef* *tag* as string or returns *default* if no *Attdef* for *tag* exist.

        Parameters
        • *tag* – name of tag
        • *default* – default value if *tag* not exist

### 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**

```python
class ezdxf.entities.dxffg.DXFGroup
```

The group name is not stored in the GROUP entity, it is stored in the `GroupCollection` object.

- **dx.dxf.description**
  - group description (string)

- **dx.dxf.unnamed**
  - 1 for unnamed, 0 for named group (int)

- **dx.dxf.selectable**
  - 1 for selectable, 0 for not selectable group (int)

- **__iter__()** → `Iterable[ezdxf.entities.dxffg.DXFGraphic]`
  - Iterate over all DXF entities in `DXFDGroup` as instances of `GraphicEntity` or inherited (LINE, CIRCLE, ...).

- **__len__()** → `int`
  - Returns the count of DXF entities in `DXFDGroup`.

- **__getitem__(item)**
  - Returns entities by standard Python indexing and slicing.

- **__contains__(item: Union[str, DXFGraphic])** → `bool`
  - Returns True if item is in `DXFDGroup`. `item` has to be a handle string or an object of type `GraphicEntity` or inherited.

- **handles()** → `Iterable[str]`
  - Iterable of handles of all DXF entities in `DXFDGroup`.

- **get_name()** → `str`
  - Get name of `DXFDGroup`.

- **edit_data()** → `List[DXFGraphic]`
  - 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[DXFGraphic])** → `None`
  - Set `entities` as new group content, should be an iterable `GraphicEntity` 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[DXFGraphic])** → `None`
  - Add `entities` to `DXFDGroup`.

- **clear()** → `None`
  - Remove all entities from `DXFDGroup`, does not delete any drawing entities referenced by this group.

- **audit(auditor: Auditor)** → `None`
  - Remove invalid handles from `DXFDGroup`.
    - 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*.

```python
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 () → Iterable[ezdxf.entities.dxfgroups.DXFGroup]
        Iterable of all existing groups.

    new (name: str = None, description: str = '', selectable: int = 1) → ezdxf.entities.dxfgroups.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 1 or not selectable if 0

        Returns:

    delete (group: Union[ezdxf.entities.dxfgroups.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
entity.dxf.layer = 'MyLayer'

# get attribute value
linetype = entity.dxf.linetype

# delete attribute
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.

Changed in version 0.10: renamed from drawing

**priority**

Integer value defining order of entities: highest value first 100 (top) before 0 (default) before -100 (bottom), priority support not implemented yet, setting priority has no effect.

**is_alive**

Returns False if entity has been deleted.

**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.

Changed in version 0.10: renamed from dxf_attrib_exists()

**is_supported_dxf_attrib(key: str) → bool**

Returns True if DXF attrib key is supported by this entity. Does not grant that attribute key really exist.

Changed in version 0.10: renamed from supports_dxf_attrib()

**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) \( \rightarrow \) None

Set new value for DXF attribute key:

```python
entity.set_dxf_attrib("layer", "MyLayer")
# same as
entity.dxf.layer = "MyLayer"
```

Raises DXFAttributeError if key is not an supported DXF attribute.

del_dxf_attrib (key: str) \( \rightarrow \) None

Delete DXF attribute key, does not raise an error if attribute is supported but not present.

Raises DXFAttributeError if key is not a supported DXF attribute.

dxfattribs (drop: Set[str] = None) \( \rightarrow \) dict

Returns a dict with all existing DXF attributes and their values and exclude all DXF attributes listed in set drop.

Changed in version 0.12: added drop argument

update_dxf_attribs (dxfattribs: dict) \( \rightarrow \) None

Set DXF attributes by a dict like `{'layer': 'test', 'color': 4}`.

set_flag_state (flag: int, state: bool = True, name: str = 'flags') \( \rightarrow \) 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.

get_flag_state (flag: int, name: str = 'flags') \( \rightarrow \) 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 () \( \rightarrow \) bool

Returns True if entity has an attached ExtensionDict.

get_extension_dict () \( \rightarrow \) ExtensionDict

Returns the existing ExtensionDict or a new created one.

has_app_data (appid: str) \( \rightarrow \) bool

Returns True if application defined data for appid exist.

get_app_data (appid: str) \( \rightarrow \) Tags

Returns application defined data for appid.

Parameters appid – application name as defined in the APPID table.

Raises DXFValueError – no data for appid found

set_app_data (appid: str, tags: Iterable) \( \rightarrow \) None

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

discard_app_data (appid: str) \( \rightarrow \) None

Discard application defined data for appid. Does not raise an exception if no data for appid exist.

has_xdata (appid: str) \( \rightarrow \) bool

Returns True if extended data for appid exist.

get_xdata (appid: str) \( \rightarrow \) 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*.

**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!

```python
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.

```
entity.rgb = (30, 40, 50)
r, g, b = entity.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`.

**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.

**zorder**

Inverted priority order (lowest value first), `zorder` support not implemented yet, setting `zorder` has no effect.

**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.

New in version 0.13.
**copy_to_layout** *(layout: BaseLayout) → DXFEntity*

Copy entity to another layout, returns new created entity as **DXFEntity** object. Copying between different DXF drawings 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 **dxfattribs** dict.

New in version 0.12.

**has_hyperlink** () → bool

Returns True if entity has an attached hyperlink.

New in version 0.12.

**get_hyperlink** () → Tuple[str, str, str]

Returns hyperlink, description and location.

New in version 0.12.

**set_hyperlink** *(link: str, description: str = None, location: str = None)*

Set hyperlink of an entity.

New in version 0.12.

**transform** *(t: Matrix44) → DXFGraphic*

Inplace transformation interface, returns **self** (floating interface).

**Parameters**
- **m** – 4x4 transformation matrix (**ezdxmath.Matrix44**)

New in version 0.13.

**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.

New in version 0.13.

**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).

New in version 0.13.

**scale_uniform** *(s: float) → DXFGraphic*

Scale entity inplace uniform about s in x-axis, y-axis and z-axis, returns **self** (floating interface).

New in version 0.13.

**rotate_x** *(angle: float) → DXFGraphic*

Rotate entity inplace about x-axis, returns **self** (floating interface).
Parameters `angle` – rotation angle in radians
New in version 0.13.

`rotate_y` *(angle: float) → DXFGraphic*
Rotate entity inplace about y-axis, returns `self` (floating interface).

Parameters `angle` – rotation angle in radians
New in version 0.13.

`rotate_z` *(angle: float) → DXFGraphic*
Rotate entity inplace about z-axis, returns `self` (floating interface).

Parameters `angle` – rotation angle in radians
New in version 0.13.

`rotate_axis` *(axis: Vector, angle: float) → DXFGraphic*
Rotate entity inplace about vector `axis`, returns `self` (floating interface).

Parameters
- `axis` – rotation axis as tuple or `Vector`
- `angle` – rotation angle in radians
New in version 0.13.

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). Smallest line weight is 13 and biggest line weight is 200, values outside this range prevents AutoCAD from loading the file. (requires DXF R2000)

  Constants defined in `ezdxf.lldxf.const`

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>

- `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, 0x020000TT 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>casts shadows</th>
<th>receives shadows</th>
<th>ignores shadows</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></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)).

Subclass of | eezxdxf.entities.DXFGraphic
DXF type    | '3DFACE'
Factory function | eezxdxf.layouts.BaseLayout.add_3dface()
Inherited DXF attributes | Common graphical DXF attributes

Warning: Do not instantiate entity classes by yourself - always use the provided factory functions!

class eezxdxf.entities.Face3d
   Face3d because 3dface is not a valid Python class name.

dxf.vtx0
   Location of 1. vertex (3D Point in WCS)
dxft1
Location of 2. vertex (3D Point in WCS)

dxft2
Location of 3. vertex (3D Point in WCS)

dxft3
Location of 4. vertex (3D Point in WCS)

dxf.invisible_edge
invisible edge flag (int, default=0)

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>first edge is invisible</td>
</tr>
<tr>
<td>2</td>
<td>second edge is invisible</td>
</tr>
<tr>
<td>4</td>
<td>third edge is invisible</td>
</tr>
<tr>
<td>8</td>
<td>fourth edge is invisible</td>
</tr>
</tbody>
</table>

Combine values by adding them, e.g. 1+4 = first and third edge is invisible.

dxf.transform (m: Matrix44) → Face3d
Transform 3DFACE entity by transformation matrix m inplace.
New in version 0.13.

**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 dxft.center and radius of dxft.radius from dxft.start_angle to dxft.end_angle. ARC goes always from dxft.start_angle to dxft.end_angle in counter clockwise orientation around the dxft.extrusion vector, which is (0, 0, 1) by default and the usual case for 2D arcs.
Subclass of: `ezdxf.entities.Circle`

DXF type: 'ARC'

Factory function: `ezdxf.layouts.BaseLayout.add_arc()`

Inherited DXF attributes: Common graphical DXF attributes

**Warning:** Do not instantiate entity classes by yourself - always use the provided factory functions!

```python
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.
    New in version 0.11.

end_point
    Returns the end point of the arc in WCS, takes OCS into account.
    New in version 0.11.

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).

transform(m: Matrix44) → Arc
    Transform ARC entity by transformation matrix m inplace.
    Raises NonUniformScalingError() for non uniform scaling.
    New in version 0.13.

to_ellipse(replace=True) → 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
    New in version 0.13.

to_spline(replace=True) → 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
    New in version 0.13.
```

5.4. Reference
Body

BODY (DXF Reference) created by an ACIS based geometry kernel provided by the Spatial Corp.
\textit{ezdx} will never interpret ACIS source code, don’t ask me for this feature.

\begin{tabular}{|l|l|}
\hline
Subclass of & \texttt{ezdx.entities.DXFGraphic} \\
DXF type & '\texttt{BODY}' \\
Factory function & \texttt{ezdx.layouts.BaseLayout.add_body()} \\
Inherited DXF attributes & \texttt{Common graphical DXF attributes} \\
Required DXF version & DXF R2000 ('AC1015') \\
\hline
\end{tabular}

\textbf{Warning:} Do not instantiate entity classes by yourself - always use the provided factory functions!

class \texttt{ezdx.entities.Body}

\begin{itemize}
\item \texttt{dxf.version}
Modeler format version number, default value is 1
\item \texttt{dxf.flags}
Require DXF R2013.
\item \texttt{dxf.uid}
Require DXF R2013.
\item \texttt{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.
\item \texttt{has\_binary\_data}
Returns \texttt{True} if ACIS data is of type \texttt{List[bytes]}, \texttt{False} if data is of type \texttt{List[str]}.
\item \texttt{tostring()} \rightarrow \texttt{str}
Returns ACIS data as one string for DXF R2000 to R2010.
\item \texttt{tobytes()} \rightarrow \texttt{bytes}
Returns ACIS data as joined bytes for DXF R2013 and later.
\item \texttt{set\_text}(text: \texttt{str}, sep: \texttt{str} = '\texttt{\n}') \rightarrow \texttt{None}
Set ACIS data from one string.
\end{itemize}

Circle

CIRCLE (DXF Reference) center at location \texttt{dxf.center} and radius of \texttt{dxf.radius}.

\begin{tabular}{|l|l|}
\hline
Subclass of & \texttt{ezdx.entities.DXFGraphic} \\
DXF type & '\texttt{CIRCLE}' \\
Factory function & \texttt{ezdx.layouts.BaseLayout.add_circle()} \\
Inherited DXF attributes & \texttt{Common graphical DXF attributes} \\
\hline
\end{tabular}

\textbf{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[Vector]
   Yields vertices of the circle for iterable angles in WCS. This method takes into account a local OCS.

   Parameters angles – iterable of angles in OCS as degrees, angle goes counter clockwise around the extrusion vector, ocs x-axis = 0 deg.

   New in version 0.11.

transform(m: Matrix44) → Circle
   Transform CIRCLE entity by transformation matrix m inplace.
   Raises NonUniformScalingError() for non uniform scaling.

   New in version 0.13.

translate(dx: float, dy: float, dz: float) → Circle
   Optimized CIRCLE/ARC translation about dx in x-axis, dy in y-axis and dz in z-axis, returns self (floating interface).

   New in version 0.13.

to_ellipse(replace=True) → 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

   New in version 0.13.

to_spline(replace=True) → 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

   New in version 0.13.

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

<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>Factory Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_linear_dim()</td>
<td>Linear and Rotated Dimension (DXF)</td>
</tr>
<tr>
<td>add_aligned_dim()</td>
<td>Aligned Dimension (DXF)</td>
</tr>
<tr>
<td>add_angular_dim()</td>
<td>Angular Dimension (DXF)</td>
</tr>
<tr>
<td>add_angular_3p_dim()</td>
<td>Angular 3P Dimension (DXF)</td>
</tr>
<tr>
<td>add_diameter_dim()</td>
<td>Diameter Dimension (DXF)</td>
</tr>
<tr>
<td>add_radius_dim()</td>
<td>Radius Dimension (DXF)</td>
</tr>
<tr>
<td>add_ordinate_dim()</td>
<td>Ordinate Dimension (DXF)</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.

- **dxf.version**
  - Version number: 0 = R2010. (int, DXF R2010)

- **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.

- **dxf.dimstyle**
  - Dimension style (`DimStyle`) name as string.

- **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 ezdxf.entities.ArcDimension 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 X-type; if not set, ordinate is Y-type</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>

- **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.

**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.

**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.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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.
### ezdxf Documentation, Release 0.13

<table>
<thead>
<tr>
<th></th>
<th>At least (taller characters will override)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Exact (taller characters will not override)</td>
</tr>
</tbody>
</table>

### `dxfl.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.

### `dxfl.actual_measurement`
Actual measurement (float, DXF R2000), this is an optional attribute and often not present. (read-only value)

### `dxfl.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.

### `dxfl.oblique_angle`
Linear dimension types with an oblique angle have an optional `dxfl.oblique_angle`.

When added to the rotation `dxfl.angle` of the linear dimension, it gives the angle of the extension lines.

### `dxfl.text_rotation`
Defines the rotation angle of the dimension text away from its default orientation (the direction of the dimension line). (float)

### `dxfl.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.

### `dimtype`
`dxfl.dimtype` without binary flags (32, 62, 128).

### `get_dim_style() → DimStyle`
Returns the associated `DimStyle` entity.

### `get_geometry_block() → Optional[BlockLayout]`
Returns `BlockLayout` of associated anonymous dimension block, which contains the entities that make up the dimension picture. Returns `None` if block name is not set or the BLOCK itself does not exist.

New in version 0.10.

### `get_measurement() → Union[float, ezdxf.math.vector.Vector]`
Returns the actual dimension measurement in 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.

New in version 0.10.2.

Changed in version 0.11: Added support for angular and ordinate dimensions.

### `override() → DimStyleOverride`
Returns the `DimStyleOverride` object.

New in version 0.12.

### `render()`
Render graphical representation as anonymous block.
New in version 0.12.

**transform** *(m: Matrix44) → Dimension*
Transform DIMENSION entity by transformation matrix m inplace.
Raises `NonUniformScalingError()` for non uniform scaling.
New in version 0.13.

**virtual_entities** *(→ Iterable[DXFGraphic]*)
Yields ‘virtual’ parts of 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.
New in version 0.12.

**explode** *(target_layout: BaseLayout = None) → EntityQuery*
Explode parts of DIMENSION as basic DXF entities like LINE, ARC or TEXT into target layout, if target
layout is None, the target layout is the layout of the DIMENSION.
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 0.12.

### DimStyleOverride

All of the `DimStyle` attributes can be overridden for each `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 ezdxr.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** *(attrs: dict) → None*

Update override dict `dimstyle_attribs`.

**Parameters**

- **attrs** – dict of DIMSTYLE attributes

**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]

**Return type**

Tuple[str, str]

**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 (name)
- **blk1** – defines left arrow as name str or as user defined block (name)
- **blk2** – defines right arrow as name str or as user defined block (name)
- **ldrblk** – defines leader arrow as name str or as user defined block (name)
- **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

```python
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 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, 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+

```python
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, linewidth: int = None, extension: float = None, disable1: bool = None, disable2: bool = None) -> None
```

Set dimension line properties

**Parameters**

- **color** – color index
- **linetype** – linetype as string
- **linewidth** – 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

```python
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

```python
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

```python
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 \((\text{location: Vertex}) \rightarrow \text{None}\)
Set text location by user, \text{location} is relative to the origin of the UCS defined in the \text{render()} method or WCS if the \text{ucs} argument is \text{None}.

\text{render} \((\text{ucs: UCS = None, discard=False}) \rightarrow \text{BaseDimensionRenderer}\)
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 \text{ezdxf}.

AutoCAD does not render DIMENSION entities automatically, so I rate AutoCAD as an unfriendly CAD application.

\text{Parameters}
- \text{ucs} – user coordinate system
- \text{discard} – discard rendering done by \text{ezdxf} (works with BricsCAD, but not with AutoCAD)

\text{Returns} Rendering object used to render the DIMENSION entity for analytics
\text{Return type} BaseDimensionRenderer

\textbf{ArcDimension}

The ARC_DIMENSION entity was introduced in DXF R2004 and is \textbf{not} documented in the DXF reference.

<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>

\textbf{Warning:} Do not instantiate entity classes by yourself - always use the provided factory functions!

\texttt{class ezdxf.entities.ArcDimension}

\texttt{\text{dx.f.\text{'ext_line1_point}}}
\texttt{\text{dx.f.\text{'ext_line2_point}}}
\texttt{\text{dx.f.\text{'arc_center}}}
\texttt{\text{dx.f.\text{'start_angle}}}
\texttt{\text{dx.f.\text{'end_angle}}}
\texttt{\text{dx.f.\text{'is_partial}}}
\texttt{\text{dx.f.\text{'has_leader}}}
\texttt{\text{dx.f.\text{'leader_point1}}}
\texttt{\text{dx.f.\text{'leader_point2}}}

5.4. Reference
**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.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th><code>ezdxf.entities.DXFGraphic</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'ELLIPSE'</td>
</tr>
<tr>
<td>factory function</td>
<td><code>add_ellipse()</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>

**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` (Vector), default value is (1, 0, 0).

`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.

New in version 0.11.

`end_point`

Returns the end point of the ellipse in WCS.

New in version 0.11.

`minor_axis`

Returns the minor axis of the ellipse as Vector in WCS.

New in version 0.12.

`construction_tool()` → `ConstructionEllipse`

Returns construction tool `ezdxf.math.ConstructionEllipse`.

New in version 0.13.

`apply_construction_tool (e: ConstructionEllipse) → Ellipse`

Set ELLIPSE data from construction tool `ezdxf.math.ConstructionEllipse`. 
New in version 0.13.

vertices(params:Iterable[float]) → Iterable[Vector]
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.

New in version 0.11.

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 ELLIPSE entity by transformation matrix m inplace.

New in version 0.13.

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).

New in version 0.13.

to_spline(replace=True) → Spline
Convert ELLIPSE 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

New in version 0.13.

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!

New in version 0.13.

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).

| Subclass of | ezdxf.entities.DXFGraphic |
| DXF type | 'HATCH' |
| Factory function | ezdxf.layouts.BaseLayout.add_hatch() |
| Inherited DXF attributes | Common graphical DXF attributes |
| Required DXF version | DXF R2000 ('AC1015') |

See also:

Tutorial for Hatch
Boundary paths helper classes

Path manager: `BoundaryPaths`

- `PolylinePath`
- `EdgePath`
  - `LineEdge`
  - `ArcEdge`
  - `EllipseEdge`
  - `SplineEdge`

Pattern and gradient helper classes

- `Pattern`
- `PatternLine`
- `Gradient`

```python
class ezdxf.entities.Hatch
```

- `dxf.pattern_name`
  Pattern name as string

- `dxf.solid_fill`
  
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>solid fill, better use: Hatch.set_solid_fill()</td>
</tr>
<tr>
<td>0</td>
<td>pattern fill, better use: Hatch.set_pattern_fill()</td>
</tr>
</tbody>
</table>

- `dxf.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.

- `dxf.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)

- `dxf.pattern_type`
dx.geom.pattern_angle
Actual pattern angle in degrees (float). Changing this value does not rotate the pattern, use set_pattern_angle() for this task.

dx.geom.pattern_scale
Actual pattern scaling factor (float). Changing this value does not scale the pattern use set_pattern_scale() for this task.

dx.geom.pattern_double
1 = double pattern size else 0. (int)

dx.geom.n_seed_points
Count of seed points (better user: get_seed_points())

dx.geom.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 range 0..255 (read/write/del)

usage:

```
color = hatch.bgcolor  # get background color as (r, g, b) tuple
hatchbgcolor = (10, 20, 30)  # set background color
del hatch.color  # delete background color
```

set_pattern_definition(lines: Sequence[T_co], factor: float = 1, angle: float = 0) → None
Setup hatch patten 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 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

New in version 0.13.

**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

New in version 0.13.

**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.

**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()`

```python
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'
- 'INVSфериCULAR'
- 'HEMISPHERICAL'
- 'INVHEMISPHERICAL'
- 'CURVED'
- 'INVCURVED'

**Parameters**

- **color1** – (r, g, b) tuple for first color, rgb values as int in range 0..255
- **color2** – (r, g, b) tuple for second color, rgb values as int in range 0..255
- **rotation** – rotation 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
set_seed_points(points: Sequence[ Tuple[float, float]]) → None
```

Set seed points, points is a list 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
transform(m: Matrix44) → Hatch
```

Transform HATCH entity by transformation matrix m inplace.

- Non uniform scaling for hatches containing arc- or ellipse edges is not correct, but at least do not produce invalid DXF files.
- New in version 0.13.

```python
associate(path: Union[PolylinePath, EdgePath], entities: Iterable[DXFEntity])
```

Set association from hatch boundary path to DXF geometry entities.

A HATCH entity can be associative to a base geometry, this association is **not** maintained nor verified by ezdxf, so if you modify the base geometry 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!
New in version 0.11.

`remove_association()`
Remove associated path elements.
New in version 0.13.

**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)

`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.

`ellipse_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 ezdxf.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 ezdxf</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. ezdxf 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 ezdxf, 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)

<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>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.

```python
class ezdxf.entities.LineEdge
    Straight boundary edge.
    start
        Start point as (x, y) tuple. (read/write)
    end
        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)
    end_angle
        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.
    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 = 0.0, base_point: Tuple[float, float] = (0.0, 0.0), offset: Tuple[float, float] = (0.0, 0.0), dash_length_items: List[float] = None) → None
        Create a new pattern definition line and add the line to the Pattern.lines attribute.
    
    static new_line (angle: float = 0.0, base_point: Tuple[float, float] = (0.0, 0.0), offset: Tuple[float, float] = (0.0, 0.0), dash_length_items: List[float] = None) → ezdxf.entities.hatch.PatternLine
        Create a new pattern definition line, but does not add the line to the Pattern.lines attribute.
    
    clear () → None
        Delete all pattern definition lines.
    
    scale (factor: float = 1, angle: float = 0) → 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
        New in version 0.13.
**class** ezdxf.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 \((\text{item} > 0 \text{ is line, } < 0 \text{ is gap, } 0.0 = \text{dot})\). (read/write)

---

**Hatch Gradient Fill Helper Classes**

**class** ezdxf.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 \((color1 \text{ mixed with white})\) 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.
### ezdxf Documentation, Release 0.13

<table>
<thead>
<tr>
<th>Subclass of</th>
<th><code>ezdxf.entities.DXFGraphic</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'IMAGE'</td>
</tr>
<tr>
<td>Factory function</td>
<td><code>ezdxf.layouts.BaseLayout.add_image()</code></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.Image
```

```python
dxf.insert
Insertion point, lower left corner of the image (3D Point in WCS).
```

```python
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
```

```python
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
```

```python
dxf.image_size
Image size in pixels as (x, y) tuple
```

```python
dxf.image_def_handle
Handle to the image definition entity, see `ImageDef`
```

```python
dxf.flags
```

<table>
<thead>
<tr>
<th><code>Image.dxf.flags</code></th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Image.SHOW_IMAGE</code></td>
<td>1</td>
<td>Show image</td>
</tr>
<tr>
<td><code>Image.SHOW_WHEN_NOT_ALIGNED</code></td>
<td>2</td>
<td>Show image when not aligned with screen</td>
</tr>
<tr>
<td><code>Image.USE_CLIPPING_BOUNDARY</code></td>
<td>4</td>
<td>Use clipping boundary</td>
</tr>
<tr>
<td><code>Image.USE_TRANSPARENCY</code></td>
<td>8</td>
<td>Transparency is on</td>
</tr>
</tbody>
</table>

```python
dxf.clipping
Clipping state:
```

<table>
<thead>
<tr>
<th>0</th>
<th>clipping off</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>clipping on</td>
</tr>
</tbody>
</table>

```python
dxf.brightness
Brightness value (0-100; default = 50)
```

```python
dxf.contrast
Contrast value (0-100; default = 50)
```

```python
dxf.fade
Fade value (0-100; default = 0)
```

```python
dxf.clipping_boundary_type
Clipping boundary type:
```
dxlf.count_boundary_points
Number of clip boundary vertices, maintained by ezdxfl.

dxfl.clip_mode
Clip mode (DXF R2010):

Outside
Inside

dxfl.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 \((\text{ImageSizeX}-0.5, \text{ImageSizeY}-0.5)\), more than two vertices is a polygon as clipping path. All vertices as pixel coordinates. (read/write)

reset_boundary_path() \rightarrow None
Reset boundary path to the default rectangle \([-0.5, -0.5), (\text{ImageSizeX}-0.5, \text{ImageSizeY}-0.5)]\).

set_boundary_path(vertices: Iterable[Vertex]) \rightarrow 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.

get_image_def() \rightarrow ImageDef
Returns the associated IMAGEDEF entity. see ImageDef.

transform(m: Matrix44) \rightarrow Image
Transform IMAGE entity by transformation matrix \(m\) inplace.

New in version 0.13.

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.

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>ezdxfl.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'LEADER'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxfl.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 ezdxfl.entities.Leader

dxf.dimstyle
Name of Dimstyle as string.
**dxflhas_arrowhead**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disabled</td>
</tr>
<tr>
<td>1</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

**dxflpath_type**

Leader path type:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Straight line segments</td>
</tr>
<tr>
<td>1</td>
<td>Spline</td>
</tr>
</tbody>
</table>

**dxflannotation_type**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Created with text annotation</td>
</tr>
<tr>
<td>1</td>
<td>Created with tolerance annotation</td>
</tr>
<tr>
<td>2</td>
<td>Created with block reference annotation</td>
</tr>
<tr>
<td>3</td>
<td>Created without any annotation (default)</td>
</tr>
</tbody>
</table>

**dxflhookline_direction**

Hook line direction flag:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Hookline (or end of tangent for a splined leader) is the opposite direction from the horizontal vector</td>
</tr>
<tr>
<td>1</td>
<td>Hookline (or end of tangent for a splined leader) is the same direction as horizontal vector (see has_hook_line)</td>
</tr>
</tbody>
</table>

**dxflhas_hookline**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No hookline</td>
</tr>
<tr>
<td>1</td>
<td>Has a hookline</td>
</tr>
</tbody>
</table>

**dxfltext_height**

Text annotation height in drawing units.

**dxfltext_width**

Text annotation width.

**dxflblock_color**

Color to use if leader’s DIMCLRd = BYBLOCK

**dxflannotation_handle**

Hard reference (handle) to associated annotation (*MText*, *Tolerance*, or *Insert* entity)

**dxflnormal_vector**

Extrusion vector? default = (0, 0, 1).

**dxflhorizontal_direction**

*Horizontal* direction for leader, default = (1, 0, 0).

**dxflleader_offset_block_ref**

Offset of last leader vertex from block reference insertion point, default = (0, 0, 0).
**dxflleader_offset_annotation_placement**

Offset of last leader vertex from annotation placement point, default = (0, 0, 0).

**vertices**

List of Vector 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 Vector.

**transform** *(m: Matrix44) -> Leader*

Transform LEADER entity by transformation matrix m inplace.

New in version 0.13.

**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**

**dxflstart**

start point of line (2D/3D Point in WCS)

**dxf.end**

dxflend
dxflend 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.

**dxfxextrusion**

extrusion vector, default value is (0, 0, 1)

**transform** *(m: Matrix44) -> Line*

Transform LINE entity by transformation matrix m inplace.

New in version 0.13.

**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, returns self (floating interface).

New in version 0.13.

**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 in OCS as \((x, y)\) tuples (\texttt{LWPolyline.dxf.elevation} is the z-axis value).

Changed in version 0.8.9: \texttt{LWPolyline} stores point data as packed data (\texttt{array.array}).

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>\texttt{ezdxf.entities.DXFGraphic}</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'LWPOLYLINE'</td>
</tr>
<tr>
<td>factory function</td>
<td>\texttt{add_lwpolyline()}</td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td>\texttt{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 \texttt{Polyline} and \texttt{LWPolyline} entities. 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

\[
\text{bulge} = 0.5
\]

\[
\text{h} = 2.5
\]

\[
\text{R} = 6.25
\]

\[
\text{bulge} = 1.0
\]

\[
10.0
\]

\[
\text{h} = 5.0
\]

\[
\text{R} = 5.0
\]
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

dxf.elevation

OCS z-axis value for all polyline points, default=0

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>

dxf.const_width

Constant line width (float), default value is 0.

dxf.count

Count of polyline points (read only), same as len(polyline)

closed

True if polyline is closed. A closed polyline has a connection from the last vertex to the first vertex. (read/write)

has_arc

Returns True if LWPOLYLINE has an arc segment.

__len__() → int

Returns count of polyline points.
__getitem__(index: int) $\rightarrow$ 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.

__setitem__(index: int, value: Sequence[float]) $\rightarrow$ 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.

__delitem__(index: int) $\rightarrow$ None

Delete point at position index, supports extended slicing.

__iter__() $\rightarrow$ Iterable[Tuple[float, float, float, float, float]]

Returns iterable of tuples (x, y, start_width, end_width, bulge).

close(state: bool = True) $\rightarrow$ None

Compatibility interface to Polyline.

vertices() $\rightarrow$ Iterable[Tuple[float, float]]

Returns iterable of all polyline points as (x, y) tuples in OCS (dx.elevation is the z-axis value).

vertices_in_wcs() $\rightarrow$ Iterable[Vertex]

Returns iterable of all polyline points as Vector(x, y, z) in WCS.

append(point: Sequence[float], format: str = 'xyseb') $\rightarrow$ 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') $\rightarrow$ 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') $\rightarrow$ 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: Matrix44) → LWPolyline
Transform LWPOLYLINE entity by transformation matrix *m* inplace.
New in version 0.13.

**virtual_entities** () → 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.
New in version 0.12.

**explode** (target_layout: BaseLayout = None) → 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.

New in version 0.12.

**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.
See also:

Tutorial for Mesh and helper classes: MeshBuilder, MeshVertexMerger

class ezdxf.entities.Mesh

    dxf.version
    dxf.blend_cres
    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)

eedit_data() → 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 MESH entity by transformation matrix m inplace.

    New in version 0.13.

MeshData

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)*
Tries 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><em>Common graphical DXF attributes</em></td>
</tr>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

```
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.dxftype</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>
```

- **dxf.flow_direction**
  Constants defined in `ezdxf.const`:
ezdxf Documentation, Release 0.13

<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>3</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>

**MText.dxf.flow_direction**

- **Value**: integer
- **Description**: text flow direction

- **MTEXT_LEFT_TO_RIGHT**: left to right
- **MTEXT_TOP_TO_BOTTOM**: top to bottom
- **MTEXT_BY_STYLE**: by style (the flow direction is inherited from the associated text style)

```plaintext
dxf.style
Text style (string); default = 'STANDARD'
dxf.text_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
dxf.line_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>

**MText.dxf.line_spacing_style**

- **Value**: integer
- **Description**: line spacing style

- **MTEXT_AT_LEAST**: taller characters will override
- **MTEXT_EXACT**: taller characters will not override

```plaintext
dxf.bg_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>

**MText.dxf.bg_fill**

- **Value**: integer
- **Description**: background fill type

- **MTEXT_BG_OFF**: no background color
- **MTEXT_BG_COLOR**: use specified color
- **MTEXT_BG_WINDOW_COLOR**: use window color
- **MTEXT_BG_CANVAS_COLOR**: use canvas background color

```plaintext
dxf.box_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 (insert: Vertex, rotation: float = None, attachment_point: int = None) → 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 (angle: float) → ezdxf.entities.mtext.MText

Set attribute `rotation` to `angle` (in degrees) and deletes `dxf.text_direction` if present.

set_bg_color (color: Union[int, str, Tuple[int, int, int], None], scale: float = 1.5)

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 (name: str, bold: bool = False, italic: bool = False, codepage: int = 1252, pitch: int = 0) → 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 (color_name: str) → None

Append text color change to existing content, `color_name` as red, yellow, green, cyan, blue, magenta or white.

add_stacked_text (upr: str, lwr: str, t: str = '^') → 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. \S1#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.

New in version 0.11.1.

**transform** (*m: Matrix44*) → `MText`

Transform MTEXT entity by transformation matrix `m` inplace.

New in version 0.13.
## 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 \SX/Y or \S1#4</td>
</tr>
<tr>
<td>\A</td>
<td>Alignment</td>
</tr>
<tr>
<td>\A0;</td>
<td>bottom</td>
</tr>
<tr>
<td>\A1;</td>
<td>center</td>
</tr>
<tr>
<td>\A2;</td>
<td>top</td>
</tr>
<tr>
<td>\C</td>
<td>Color change</td>
</tr>
<tr>
<td>\C1;</td>
<td>red</td>
</tr>
<tr>
<td>\C2;</td>
<td>yellow</td>
</tr>
<tr>
<td>\C3;</td>
<td>green</td>
</tr>
<tr>
<td>\C4;</td>
<td>cyan</td>
</tr>
<tr>
<td>\C5;</td>
<td>blue</td>
</tr>
<tr>
<td>\C6;</td>
<td>magenta</td>
</tr>
<tr>
<td>\C7;</td>
<td>white</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.point`.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th><code>ezdxf.entities.DXFGraphic</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'POINT'</td>
</tr>
<tr>
<td>Factory function</td>
<td><code>ezdxf.layouts.BaseLayout.add_point()</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.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 POINT entity by transformation matrix `m` inplace.

  New in version 0.13.

- `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, returns `self` (floating interface).

  New in version 0.13.
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>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'POLYLINE'</td>
</tr>
<tr>
<td>2D factory function</td>
<td>ezdxf.layouts.BaseLayout.add_polyline2d()</td>
</tr>
<tr>
<td>3D factory function</td>
<td>ezdxf.layouts.BaseLayout.add_polyline3d()</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.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`:
### Polyline.dxf.flags

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLYLINE_CLOSED</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 equals POLYLINE_CLOSED</td>
</tr>
<tr>
<td>POLYLINE_MESH_CLOSED_N_DIRECTION</td>
<td>32 The polygon mesh is closed in the N direction</td>
</tr>
<tr>
<td>POLYLINE_CURVE_FIT_VERTICES_ADDED</td>
<td>2 Curve-fit vertices have been added</td>
</tr>
<tr>
<td>POLYLINE_SPLINE_FIT_VERTICES_ADDED</td>
<td>4 Spline-fit vertices have been added</td>
</tr>
<tr>
<td>POLYLINE_3D_POLYLINE</td>
<td>8 This is a 3D Polyline</td>
</tr>
<tr>
<td>POLYLINE_3D_POLYMESH</td>
<td>16 This is a 3D polygon mesh</td>
</tr>
<tr>
<td>POLYLINE_MESH_CLOSED_N_DIRECTION</td>
<td>64 This Polyline is a polyface mesh</td>
</tr>
<tr>
<td>POLYLINE_POLYFACE_MESH</td>
<td>128 The linetype pattern is generated continuously around the vertices of this Polyline</td>
</tr>
<tr>
<td>POLYLINE_GENERATE_LINETYPE_PATTERN</td>
<td></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

**dxf.m_count**

Polymesh M vertex count (int); default = 1

**dxf.n_count**

Polymesh N vertex count (int); default = 1

**dxf.m_smooth_density**

Smooth surface M density (int); default = 0

**dxf.n_smooth_density**

Smooth surface N density (int); default = 0

**dxf.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.

**get_mode** () → str
Returns a string: 'AcDb2dPolyline', 'AcDb3dPolyline', 'AcDbPolygonMesh' or '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.Vector]
Returns iterable of all polyline vertices as (x, y, z) tuples, not as `Vertex` objects.

**append_vertex**(point: Vertex, dxfattribs: dict = None) → None
Append 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 = 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 = None) → None
Append multiple `Vertex` entities at location `points`.

Parameters
• points – iterable of (x, y[, start_width, [end_width, [bulge]]]) tuple
• format – format: format string, default is 'xy', see: `User Defined Point Format Codes`
• `dxfattribs` – dict of DXF attributes for `Vertex` class

```python
insert_vertices(pos: int, points: Iterable[Vertex], dxfattribs: dict = None) → None
```
Insert `Vertex` entities at location `points` at insertion position `pos` of list `Polyline.vertices`.

**Parameters**

- `pos` – insertion position of list `Polyline.vertices`
- `points` – list of `(x, y[, z])` tuples
- `dxfattribs` – dict of DXF attributes for `Vertex` class

```python
transform(m: Matrix44) → Polyline
```
Transform POLYLINE entity by transformation matrix `m` inplace.

New in version 0.13.

```python
virtual_entities() → Iterable[Union[Line, Arc]]
```
Yields ‘virtual’ parts of POLYLINE as LINE, ARC or 3DFACE 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.

New in version 0.12.

```python
explode(target_layout: BaseLayout = None) → EntityQuery
```
Explode parts of POLYLINE as LINE, ARC or 3DFACE entities into target layout, if target layout is `None`, the target layout is the layout of the POLYLINE.

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 0.12.

**Vertex**

A VERTEX (VERTEX DXF Reference) represents a polyline/mesh vertex.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th><code>ezdxf.entities.DXFGraphic</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'VERTEX'</td>
</tr>
<tr>
<td>Factory function</td>
<td><code>Polyline.append_vertex()</code></td>
</tr>
<tr>
<td>Factory function</td>
<td><code>Polyline.extend()</code></td>
</tr>
<tr>
<td>Factory function</td>
<td><code>Polyline.insert_vertices()</code></td>
</tr>
<tr>
<td>Inherited DXF Attributes</td>
<td><code>Common graphical DXF attributes</code></td>
</tr>
</tbody>
</table>

```python
class ezdxf.entities.Vertex
```

```python
dxf.location
```
Vertex location (2D/3D Point `OCS` when 2D, `WCS` when 3D)

```python
dxf.start_width
```
Line segment start width (float); default = 0

```python
dxf.end_width
```
Line segment end width (float); default = 0
dxfr.bulge

Bulge value (float); default = 0.

The bulge value is used to create arc shaped line segments.

dxfr.flags

Constants defined in ezdxf.lldxf.const:

<table>
<thead>
<tr>
<th>Vertex.dxf.flags</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VTX_EXTRA_VERTEX_CREATED</td>
<td>1</td>
<td>Extra vertex created by curve-fitting</td>
</tr>
<tr>
<td>VTX_CURVE_FIT_TANGENT</td>
<td>2</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_CREATED</td>
<td>8</td>
<td>Spline vertex created by spline-fitting</td>
</tr>
<tr>
<td>VTX_SPLINE_FRAME_CONTROL_POINT</td>
<td>16</td>
<td>Spline frame control point</td>
</tr>
<tr>
<td>VTX_3D_POLYLINE_VERTEX</td>
<td>32</td>
<td>3D polyline vertex</td>
</tr>
<tr>
<td>VTX_3D_POLYGON_MESH_VERTEX</td>
<td>64</td>
<td>3D polygon mesh vertex</td>
</tr>
<tr>
<td>VTX_3D_POLYFACE_MESH_VERTEX</td>
<td>128</td>
<td>Polyface mesh vertex</td>
</tr>
</tbody>
</table>

dxfr.tangent

Curve fit tangent direction (float), used for 2D spline in DXF R12.

dxfr.vtx1

Index of 1st vertex, if used as face (feature for experts)

dxfr.vtx2

Index of 2nd vertex, if used as face (feature for experts)

dxfr.vtx3

Index of 3rd vertex, if used as face (feature for experts)

dxfr.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

Polymesh

Subclass of ezdxf.entities.Polyline

DXF type 'POLYLINE'

Factory function ezdxf.layouts.BaseLayout.add_polymesh()

Inherited DXF Attributes Common graphical DXF attributes

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'.

5.4. 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'.

226 Chapter 5. Contents
**append_face** *(face: FaceType, dxfattrs: dict = None) → None*

Append a single face. A face is a list of \((x, y, z)\) tuples.

**Parameters**

- **face** – List\([(x, y, z)]\)
- **dxfattrs** – dict of DXF attributes for Vertex entity

**append_faces** *(faces: Iterable[FaceType], dxfattrs: dict = 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\([(x, y, z)]\)
- **dxfattrs** – 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 with vertex optimization. Merges vertices with nearly same vertex locations. Polyfaces created by ezdxf are optimized automatically.

**Parameters** precision – decimal 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 XLINE/RAY entity by transformation matrix \(m\) inplace.

  New in version 0.13.

- **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, returns self (floating interface).

  New in version 0.13.
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><code>.ezdxf.entities.Body</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'REGION'</td>
</tr>
<tr>
<td>Factory function</td>
<td><code>.ezdxf.layouts.BaseLayout.add_region()</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>

**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><code>.ezdxf.entities.DXFGraphic</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'SHAPE'</td>
</tr>
<tr>
<td>Factory function</td>
<td><code>.ezdxf.layouts.BaseLayout.add_shape()</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.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: Matrix44) → Shape*

Transform SHAPE entity by transformation matrix *m* inplace.

New in version 0.13.

**Solid**

SOLID *(DXF Reference)* is a 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>'SOLID'</td>
</tr>
<tr>
<td>Factory function</td>
<td>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!

**class ezdxf.entities.Solid**

dxf.vtx0
Location of 1. vertex (2D/3D Point in *OCS*)

dxf.vtx1
Location of 2. vertex (2D/3D Point in *OCS*)

dxf.vtx2
Location of 3. vertex (2D/3D Point in *OCS*)

dxf.vtx3
Location of 4. vertex (2D/3D Point in *OCS*)

**transform** *(m: Matrix44) → Solid*

Transform SOLID/TRACE entity by transformation matrix *m* inplace.

New in version 0.13.

**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.

**See also:**
- Wikipedia article about B_splines
- Department of Computer Science and Technology at the Cambridge University
- Tutorial for Spline
Since ezdxf v0.8.9 Spline stores fit- and control points, knots and weights 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>'SPLINE'</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>
<tr>
<td>Required DXF version</td>
<td>DXF R2000 ('AC1015')</td>
</tr>
</tbody>
</table>

### Factory Functions

<table>
<thead>
<tr>
<th>Basic spline entity</th>
<th>add_spline()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spline control frame from fit points</td>
<td>add_spline_control_frame()</td>
</tr>
<tr>
<td>Open uniform spline</td>
<td>add_open_spline()</td>
</tr>
<tr>
<td>Closed uniform spline</td>
<td>add_closed_spline()</td>
</tr>
<tr>
<td>Open rational uniform spline</td>
<td>add_rational_spline()</td>
</tr>
<tr>
<td>Closed rational uniform spline</td>
<td>add_closed_rational_spline()</td>
</tr>
</tbody>
</table>

```python
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
- **dxf.fit_tolerance**: Fit tolerance (float); default = 1e-10
- **dxf.control_point_tolerance**: Control point tolerance (float); default = 1e-10
- **dxf.start_tangent**: Start tangent vector as (3D vector in WCS)
**dxf.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 construction tool `ezdxf.math.BSpline`.

New in version 0.13.

**apply_construction_tool**(s: BSpline) → Spline
Set SPLINE data from construction tool `ezdxf.math.BSpline` or from a `geomdl.BSpline`. Curve object.

New in version 0.13.

**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.

**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, deos 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.

5.4. Reference
**transform** \((m: \text{Matrix44}) \rightarrow \text{Spline}\)  
Transform SPLINE entity by transformation matrix \(m\) inplace.  
New in version 0.13.

**classmethod from_arc** \((\text{entity: DXFGraphic}) \rightarrow \text{Spline}\)  
Create a new SPLINE entity from 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!  
New in version 0.13.

### 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><code>ezdxf.entities.Body</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'SURFACE'</td>
</tr>
<tr>
<td>Factory function</td>
<td><code>ezdxf.layouts.BaseLayout.add_surface()</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>

**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`.  

 dx.f.u_count  
Number of U isolines.  

 dx.f.v_count  
Number of V2 isolines.

### ExtrudedSurface

(DXF Reference)

<table>
<thead>
<tr>
<th>Subclass of</th>
<th><code>ezdxf.entities.Surface</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'EXTRUDEDSURFACE'</td>
</tr>
<tr>
<td>Factory function</td>
<td><code>ezdxf.layouts.BaseLayout.add_extruded_surface()</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 R2007 ('AC1021')</td>
</tr>
</tbody>
</table>

**class** `ezdxf.entities.ExtrudedSurface`  
Same attributes and methods as parent class `Surface`.  

 dx.f.class_id  

dx.f.sweep_vector  

dx.f.draft_angle
LoftedSurface

(DFX Reference)

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.Surface</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFX type</td>
<td>'LOFTEDSURFACE'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_lofted_surface()</td>
</tr>
<tr>
<td>Inherited DFX attributes</td>
<td>Common graphical DFX attributes</td>
</tr>
<tr>
<td>Required DFX version</td>
<td>DXF R2007 ('AC1021')</td>
</tr>
</tbody>
</table>

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
RevolvedSurface

(DXF Reference)

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.Surface</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'REVOLVEDSURFACE'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_revolved_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>

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

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>

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
transformation_matrix_path_entity()
type: Matrix44

sweep_entity_transformation_matrix()
type: Matrix44
path_entity_transformation_matrix()
type: Matrix44

Text

One line TEXT entity (DXF Reference). Text.dxf.height in drawing units and defaults to 1, but it also depends
on the font rendering of the CAD application. Text.dxf.width is a scaling factor, but the DXF reference does not
define the base value to scale, in practice the Text.dxf.height is the base value, the effective text width depends
on the font defined by `Text.dxf.style` and the font rendering of the CAD application, especially for proportional fonts, text width calculation is nearly impossible without knowledge of the used CAD application and their font rendering behavior. This is one reason why the DXF and also DWG file format are not reliable for exchanging exact text layout, they are just reliable for exchanging exact geometry.

See also:

* Tutorial for Text

<table>
<thead>
<tr>
<th>Subclass of</th>
<th><code>ezdxf.entities.DXFGraphic</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'TEXT'</td>
</tr>
<tr>
<td>Factory function</td>
<td><code>ezdxf.layouts.BaseLayout.add_text()</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.Text`

  * `dxf.text` Text content. (str)
  * `dxf.insert` First alignment point of text (2D/3D Point in `OCS`), relevant for the adjustments 'LEFT', 'ALIGN' and 'FIT'.
  * `dxf.align_point` Second alignment point of text (2D/3D Point in `OCS`), if the justification is anything other than 'LEFT', the second alignment point specify also the first alignment point: (or just the second alignment point for 'ALIGN' and 'FIT')
  * `dxf.height` Text height in drawing units (float); default value is 1
  * `dxf.rotation` Text rotation in degrees (float); default value is 0
  * `dxf.oblique` Text oblique angle in degrees (float); default value is 0 (straight vertical text)
  * `dxf.style` `Textstyle` name (str); default value is 'Standard'
  * `dxf.width` Width scale factor (float); default value is 1
  * `dxf.halign` Horizontal alignment flag (int), use `set_pos()` and `get_align()`; default value is 0

<table>
<thead>
<tr>
<th><code>halign</code></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 (int), use `set_pos()` and `get_align()`; default value is 0

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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 (int)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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 has vertical alignment Baseline.

- '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, but the result is the same as for 'MIDDLE_CENTER'.

**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()** → Tuple[str, Vertex, Optional[Vertex]]

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()** → str

Returns the actual text alignment as string, see also `set_pos()`.

**set_align**(align: str = 'LEFT') → Text

Just for experts: Sets the text alignment without setting the alignment points, set adjustment points `attr:dxr.insert` and `dxr.align_point` manually.

**Parameters**

- align – test alignment, see also `set_pos()`
**transform** (*m*: *Matrix44*) → *Text*
Transform TEXT entity by transformation matrix *m* inplace.
New in version 0.13.

**translate** (*dx*: *float*, *dy*: *float*, *dz*: *float*) → *Text*
Optimized TEXT/ATTRIB/ATTDEF translation about *dx* in x-axis, *dy* in y-axis and *dz* in z-axis, returns *self* (floating interface).
New in version 0.13.

**plain_text** () → *str*
Returns text content without formatting codes.
New in version 0.13.

---

**Trace**

TRACE entity (**DXF Reference**) is 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)**) . I don’t know the difference between SOLID and TRACE.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th><strong>ezdxf.entities.DXFGraphic</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'TRACE'</td>
</tr>
<tr>
<td>Factory function</td>
<td><strong>ezdxf.layouts.BaseLayout.add_trace()</strong></td>
</tr>
<tr>
<td>Inherited DXF attributes</td>
<td><strong>Common graphical DXF attributes</strong></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 *OCS*)

- **dxf.vtx1**
  Location of 2. vertex (2D/3D Point in *OCS*)

- **dxf.vtx2**
  Location of 3. vertex (2D/3D Point in *OCS*)

- **dxf.vtx3**
  Location of 4. vertex (2D/3D Point in *OCS*)

**transform** (*m*: *Matrix44*) → *Trace*
Transform SOLID/TRACE entity by transformation matrix *m* inplace.
New in version 0.13.

---

**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 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.

underlay_def
   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 in modelspace (3D point in WCS).

dxf.width
    Viewport width in paperspace units (float)

dxf.height
    Viewport height in paperspace units (float)

dxf.status
    Viewport status field (int)

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>On, but is fully off screen, or is one of the viewports that is not active because the $MAXACTVP count is currently being exceeded.</td>
</tr>
<tr>
<td>0</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 DCS.

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
**dxv.view_height**
View height in WCS.

**dxv.view_twist_angle**

**dxv.circle_zoom**

**dxv.flags**
Viewport status bit-coded flags:

<table>
<thead>
<tr>
<th>Value (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>kIsoPairTop. If set and kIsoPairRight is not set, then isopair top is enabled. If both kIsoPairTop and kIsoPairRight are set, then isopair left is enabled</td>
</tr>
<tr>
<td>8192 (0x2000)</td>
<td>kIsoPairRight. If set and kIsoPairTop 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>

**dxv.clipping_boundary_handle**

**dxv.plot_style_name**

**dxv.render_mode**
0 2D Optimized (classic 2D)
1 Wireframe
2 Hidden line
3 Flat shaded
4 Gouraud shaded
5 Flat shaded with wireframe
6 Gouraud shaded with wireframe

dxf.ucs_per_viewport

dxf.ucs_icon

dxf.ucs_origin
    UCS origin as 3D point.

dxf.ucs_x_axis
    UCS x-axis as 3D vector.

dxf.ucs_y_axis
    UCS y-axis as 3D vector.

dxf.ucs_handle
    Handle of UCSTable if UCS is a named UCS. If not present, then UCS is unnamed.

dxf.ucs_ortho_type

<table>
<thead>
<tr>
<th>0</th>
<th>not orthographic</th>
</tr>
</thead>
<tbody>
<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.ucs_base_handle
    Handle of UCSTable of base UCS if UCS is orthographic (Viewport.dxf.ucs_ortho_type is non-zero). If not present and Viewport.dxf.ucs_ortho_type is non-zero, then base UCS is taken to be WORLD.

dxf.elevation

dxf.shade_plot_mode
    (DXF R2004)

<table>
<thead>
<tr>
<th>0</th>
<th>As Displayed</th>
</tr>
</thead>
<tbody>
<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>

dxf.grid_frequency
    Frequency of major grid lines compared to minor grid lines. (DXF R2007)

dxf.background_handle

dxf.shade_plot_handle
ezdxf Documentation, Release 0.13

dxf.visual_style_handle
dxf.default_lighting_flag
dxf.default_lighting_style

<table>
<thead>
<tr>
<th></th>
<th>One distant light</th>
<th>Two distant lights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dxf.view_brightness
dxf.view_contrast
dxf.ambient_light_color_1
   as AutoCAD Color Index (ACI)
dxf.ambient_light_color_2
   as true color value
dxf.ambient_light_color_3
   as true color value
dxf.sun_handle
dxf.ref_vp_object_1
dxf.ref_vp_object_2
dxf.ref_vp_object_3
dxf.ref_vp_object_4

frozen_layers
   Set/get frozen layers as list of layer names.

XLine

XLINE entity (DXF Reference) is a construction line that extents to infinity in both directions.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxf.entities.DXFGraphic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'XLINE'</td>
</tr>
<tr>
<td>Factory function</td>
<td>ezdxf.layouts.BaseLayout.add_xline()</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.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 XLINE/RAY entity by transformation matrix m inplace.
   New in version 0.13.
**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, returns *self* (floating interface).

New in version 0.13.

### DXF Objects

#### DXFObject

Common base class for all non-graphical DXF objects.

```python
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!

```python
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 *ezdx*):

<table>
<thead>
<tr>
<th>0</th>
<th>not applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>keep existing</td>
</tr>
<tr>
<td>2</td>
<td>use clone</td>
</tr>
<tr>
<td>3</td>
<td>$\text{&lt;xref&gt;}$D$\text{&lt;name&gt;}$</td>
</tr>
<tr>
<td>4</td>
<td>$\text{D}$D$\text{&lt;name&gt;}$</td>
</tr>
<tr>
<td>5</td>
<td>Unmangle name</td>
</tr>
</tbody>
</table>

**is_hard_owner**

True if **Dictionary** is hard owner of entities. Hard owned entities will be deleted by deleting the dictionary.

5.4. Reference
__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 DXFDictionary, 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 exit.

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><code>ezdxf.entities.Dictionary</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'ACDBDICTIONARYWDFLT'</td>
</tr>
<tr>
<td>Factory function</td>
<td><code>ezdxf.sections.objects.ObjectsSection.add_dictionary_with_default()</code></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.

    set_default(default) → None
    Set dictionary wide default entry.

    Parameters default – default entry as hex string or as DXFEntity
```

## DictionaryVar

<table>
<thead>
<tr>
<th>Subclass of</th>
<th><code>ezdxf.entities.DXFObject</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'DICTIONARYVAR'</td>
</tr>
<tr>
<td>Factory function</td>
<td><code>ezdxf.entities.Dictionary.add_dict_var()</code></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.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th><code>ezdxf.entities.DXFObject</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'GEODATA'</td>
</tr>
<tr>
<td>Factory function</td>
<td><code>ezdxf.layouts.Modelspace.new_geodata()</code></td>
</tr>
</tbody>
</table>

#### Required DXF version
R2010('AC1024')

See also:
using_geodata.py

### Warning:
Do not instantiate object classes by yourself - always use the provided factory functions!

```python
class ezdxf.entities.GeoData
```
ezdxf Documentation, Release 0.13

**dxf.version**

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R2009</td>
</tr>
<tr>
<td>2</td>
<td>R2010</td>
</tr>
</tbody>
</table>

**dxf.coordinate_type**

<table>
<thead>
<tr>
<th>Coordinate Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>unknown</td>
</tr>
<tr>
<td>1</td>
<td>local grid</td>
</tr>
<tr>
<td>2</td>
<td>projected grid</td>
</tr>
<tr>
<td>3</td>
<td>geographic (latitude/longitude)</td>
</tr>
</tbody>
</table>

**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 coordinate system coordinates, valid only when coordinate type is **local grid**.

**dxf.north_direction**

North direction as 2D vector.

**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 per UnitsValue enumeration. Will be kUnitsUndefined if units specified by horizontal unit scale is not supported by AutoCAD enumeration.

**dxf.vertical_units**

Vertical units per UnitsValue enumeration. Will be kUnitsUndefined if units specified by vertical unit scale is not supported by AutoCAD enumeration.

**dxf.up_direction**

Up direction as 3D vector.

**dxf.scale_estimation_method**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</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>prismoidal</td>
</tr>
</tbody>
</table>

**dxf.sea_level_correction**

Bool flag specifying whether to do sea level correction.

**dxf.user_scale_factor**
dxfs.sea_level_elevation  
dxfs.coordinate_projection_radius  
dxfs.geo rss_tag  
dxfs.observation_from_tag  
dxfs.observation_to_tag  
dxfs.mesh_faces_count

source_vertices  
2D source vertices as VertexArray.

target_vertices  
2D target vertices as VertexArray.

faces  
List of face definition tuples, each face entry is a 3-tuple of vertex indices (0-based).

cordinate_system_definition  
The coordinate system definition string. (Always a XML string?)

ImageDef

IMAGDEF entity defines an image file, which can be placed by the Image entity.

<table>
<thead>
<tr>
<th>Subclass of</th>
<th>ezdxfs.entities.DXFObj</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>'IMAGDEF'</td>
<td></td>
</tr>
<tr>
<td>Factory function (1)</td>
<td>ezdxfs.drawing.Drawing.add_image_def()</td>
<td></td>
</tr>
<tr>
<td>Factory function (2)</td>
<td>ezdxfs.sections.objects.ObjectsSection.add_image_def()</td>
<td></td>
</tr>
</tbody>
</table>

Warning: Do not instantiate object classes by yourself - always use the provided factory functions!

class ezdxfs.entities.ImageDef

dxfs.class_version  
Current version is 0.

dxfs.filename  
Relative (to the DXF file) or absolute path to the image file as string.

dxfs.image_size  
Image size in pixel as (x, y) tuple.

dxfs.pixel_size  
Default size of one pixel in drawing units as (x, y) tuple.

dxfs.loaded  
0 = unloaded; 1 = loaded, default = 1

dxfs.resolution_units
ImageDefReactor

class ezdxf.entities.ImageDefReactor

    dxf.class_version
    dxf.image_handle

DXFLayout

LAYOUT entity is part of a modelspace or paperspace layout definitions.

| Subclass of | ezdxf.entities.PlotSettings |
| DXF type    | 'LAYOUT'                    |
| Factory function | internal data structure, use Layouts to manage layout objects. |

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!

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.DXObject`
---|---
DXF type | 'PLOTSETTINGS'
Factory function | internal data structure

```python
class ezdxf.entities.PlotSettings
    dxf.page_setup_name
        Page setup name

TODO
```

**Sun**

▲ S▲N entity defines properties of the sun.

```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
        0  Sun do not cast shadows
        1  Sun do cast shadows
    dxf.shadow_type
    dxf.shadow_map_size
    dxf.shadow_softness
```

5.4. Reference
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.DXObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXF type</td>
<td>internal base class</td>
</tr>
<tr>
<td>Factory function (1)</td>
<td>ezdxf.drawing.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.

<table>
<thead>
<tr>
<th>'pdf'</th>
<th>PDF page number</th>
</tr>
</thead>
<tbody>
<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.drawing.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.drawing.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.drawing.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.

5.4.2 Data Query

See also:

For usage of the query features see the tutorial: Tutorial for getting data from DXF files
Entity Query String

Entity Query String := 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 an 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[DXFEntity]

Returns iterable of DXFEntity objects.

extend(entities: Iterable[DXFEntity], query: str = '*', unique: bool = True) → 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: str = '*') → ezdxf.query.EntityQuery

Returns a new EntityQuery container with all entities matching this additional query.

raises: ParseException (pyparsing.py)

groupby(dxattrib: str = ", key: Callable[[DXFEntity], Hashable] = None) → Dict[Hashable, List[DXFEntity]]

Returns a dict of entity lists, where entities are grouped by a DXF attribute or a key function.

Parameters

• *dxfattrib* – grouping DXF attribute as string like 'layer'

• *key* – key function, which accepts a DXFEntity as argument, returns grouping key of this entity or None for ignore this object. Reason for ignoring: a queried DXF attribute is not supported by this entity

**The new() Function**

ezdxf.query.new(entities: Iterable['DXFEntity'] = None, query: str = '*') → EntityQuery

Start a new query based on sequence entities. The entities argument has to be an iterable of DXFEntity or inherited objects and returns an EntityQuery object.

See also:

For usage of the groupby features see the tutorial: Retrieve entities by groupby() function

---

5.4. Reference
Groupby Function

ezdx��.groupby

```python
def group_key(entity: DXFEntity):
    return entity.dxf.layer, entity.dxf.color
```

For not suitable DXF entities return None to exclude this entity, in this case it’s not required, because groupby() catches DXFAttributeError exceptions to exclude entities, which do not provide layer and/or color attributes, automatically.

Result dict for `dxfattrib = 'layer'` may look like this:

```python
{
    '0': [ ... list of entities ],
    'ExampleLayer1': [ ... ],
    'ExampleLayer2': [ ... ],
    ...
}
```

Result dict for `key = group_key`, which returns a (layer, color) tuple, may look like this:

```python
{
    ('0', 1): [ ... list of entities ],
    ('0', 3): [ ... ],
    ('0', 7): [ ... ],
    ('ExampleLayer1', 1): [ ... ],
    ('ExampleLayer1', 2): [ ... ],
    ('ExampleLayer1', 5): [ ... ],
    ('ExampleLayer2', 7): [ ... ],
    ...
}
```

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.

5.4.3 Math Utilities

Utility functions and classes located in module `ezdx��.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 Vector compatible object
• p2 – second vertex as Vector compatible object
• abs_tol – absolute tolerance

Raises TypeError – for incompatible vertices

ezdxf.math.closest_point(base: Vertex, points: Iterable[Vertex]) → Vector

Returns closest point to base.

Parameters

• base – base point as Vector compatible object
• points – iterable of points as Vector 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.open_uniform_knot_vector(count: int, order: int, normalize=False) → List[float]

Returns an open (clamped) 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.required_knot_values(count: int, order: int) → int

Returns the count of required knot values for a B-spline of order and count control points.

Parameters

• count – count of control points, in text-books referred as “n + 1”
• order – order of B-Spline, in text-books referred as “k”

Relationship:

“p” is the degree of the B-spline, text-book notation.

• k = p + 1
• 2 k n + 1
ezdxf.math.xround(value: float, rounding: float = 0.0) → float

Extended rounding function, argument rounding defines the rounding limit:

<table>
<thead>
<tr>
<th>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 x.1, x.2, ... x.0</td>
</tr>
<tr>
<td>0.25</td>
<td>round next to x.25, x.50, x.75 or x.00</td>
</tr>
<tr>
<td>0.5</td>
<td>round next to 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: xxx2, xxx4, xxx6 ...</td>
</tr>
<tr>
<td>5.0</td>
<td>round to a multiple of 5: xxx5 or xxx0</td>
</tr>
<tr>
<td>10.0</td>
<td>round to a multiple of 10: xx10, xx20, ...</td>
</tr>
</tbody>
</table>

Parameters

- **value** – float value to round
- **rounding** – rounding limit

ezdxf.math.linspace(start: float, stop: float, num: int, endpoint=True) → Iterable[float]

Return evenly spaced numbers over a specified interval, like numpy.linspace().

Returns num evenly spaced samples, calculated over the interval [start, stop]. The endpoint of the interval can optionally be excluded.

New in version 0.12.3.

Bulge Related Functions

See also:

Description of the Bulge value.

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

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

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

ezdxf.math.bulge_to_arc(start_point: Vertex, end_point: Vertex, bulge: float) → Tuple[Vec2, float, float, float]

Returns arc parameters from bulge parameters.
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

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

ezdxf.math.distance_point_line_2d(point: Vec2, start: Vec2, end: Vec2) → float

Returns distance from point to line defined by start- and end point.

**Parameters**

- **point** – 2D point to test as `Vec2` or tuple of float
- **start** – line definition point as `Vec2` or tuple of float
- **end** – line definition point as `Vec2` or tuple of float

New in version 0.11.
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 `{x,y}`
• **end** – line definition point as `{x,y}`
• **abs_tol** – tolerance for minimum distance to line

New in version 0.11.

```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 `{x,y}`
• **start** – line definition point as `{x,y}`
• **end** – line definition point as `{x,y}`
• **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

New in version 0.11.

```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 `{x,y}`
• **start** – line definition point as `{x,y}`
• **end** – line definition point as `{x,y}`
• **colinear** – a colinear point is also “left of line” if True

Changed in version 0.11: renamed from `is_left_of_line()`

```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 `{x,y}`
• **polygon** – iterable of 2D points as `{x,y}`
• **abs_tol** – tolerance for distance check

Returns +1 for inside, 0 for on boundary line, -1 for outside

New in version 0.11.

```python
ezdxf.math.convex_hull_2d(points: Iterable[Vertex]) -> List[Vertex]
```

Returns 2D convex hull for `points`.

**Parameters**

• **points** – iterable of points as `{x,y}`

```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

New in version 0.11.

ezdxf.math.rytz_axis_construction (d1: Vector, d2: Vector) \(\rightarrow\) Tuple[Vector, Vector, 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.

**Source:** Wikipedia

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 Vector

• **d2** – conjugated semi-minor axis as Vector

**Returns** Tuple of (major axis, minor axis, ratio)

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.

New in version 0.11.

**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

\texttt{ezdxf.math.normal_vector_3p(a: Vector, b: Vector, c: Vector) \rightarrow Vector}

Returns normal vector for 3 points, which is the normalized cross product for: \(a \rightarrow b \times a \rightarrow c\).

New in version 0.11.

\texttt{ezdxf.math.is_planar_face(face: Sequence[Vector], abs_tol=1e-9) \rightarrow bool}

Returns \texttt{True} if sequence of vectors is a planar face.

**Parameters**

- \texttt{face} – sequence of \texttt{Vector} objects
- \texttt{abs_tol} – tolerance for normals check

New in version 0.11.

\texttt{ezdxf.math.subdivide_face(face: Sequence[Union[Vector, Vec2]], quads=True) \rightarrow Iterable[List[Vector]]}

Yields new subdivided faces. Creates new faces from subdivided edges and the face midpoint by linear interpolation.

**Parameters**

- \texttt{face} – a sequence of vertices, \texttt{Vec2} and \texttt{Vector} objects supported.
- \texttt{quads} – create quad faces if \texttt{True} else create triangles

New in version 0.11.

\texttt{ezdxf.math.subdivide_ngons(faces: Iterable[Sequence[Union[Vector, Vec2]]]) \rightarrow Iterable[List[Vector]]}

Yields only triangles or quad faces, subdivides ngons into triangles.

**Parameters**

- \texttt{faces} – iterable of faces as sequence of \texttt{Vec2} and \texttt{Vector} objects

New in version 0.12.

\texttt{ezdxf.math.intersection_ray_ray_3d(ray1: Tuple[Vector, Vector], ray2: Tuple[Vector, Vector], abs_tol=1e-10) \rightarrow Sequence[Vector]}

Calculate intersection of two 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 on the ray as `Vector` objects
• **ray2** – second ray as tuple of two points on the ray as `Vector` objects
• **abs_tol** – absolute tolerance for comparisons

New in version 0.11.

```python
ezdxf.math.estimate_tangents(points: List[Vector], method: str = '5-points', normalize = True) → List[Vector]
```

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 `Vector` objects

```python
ezdxf.math.estimate_end_tangent_magnitude(points: List[Vector], method: str = 'chord') → List[Vector]
```

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

```python
ezdxf.math.fit_points_to_cad_cv(fit_points: Iterable[Vertex], degree: int = 3, method='chord', tangents: Iterable[Vertex] = None) → BSpline
```

Returns the control vertices and knot vector configuration for DXF SPLINE entities defined only by 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 limited to 2 or 3, a stored degree of >3 is ignored, this limit exist only 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.

**Parameters**

- `fit_points` – points the spline is passing through
- `degree` – degree of spline, only 2 or 3 is supported by BricsCAD, default = 3
- `method` – knot parametrization method, default = ‘chord’
- `tangents` – start- and end tangent, default is autodetect

**Returns** `BSpline`

New in version 0.13.

```python
ezdxf.math.global_bspline_interpolation(fit_points: Iterable[Vertex], degree: int = 3, tangents: Iterable[Vertex] = None, method: str = 'chord') -> BSpline
```

B-spline interpolation by 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 `Vector` 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`

```python
ezdxf.math.local_cubic_bspline_interpolation(fit_points: Iterable[Vertex], method: str = '5-points', tangents: Iterable[Vertex] = None) -> 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 Vector compatible objects
• method – tangent estimation method
• tangents – tangents as Vector compatible objects (optional)

Returns BSpline

ezdx.f.math.rational_spline_from_arc(center: Vector = (0, 0), radius:float=1, start_angle: float = 0, end_angle: float = 360, segments: int = 1) → BSpline

Returns a rational B-splines for a circular 2D arc.

Parameters

• center – circle center as Vector 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), 1 for as few as needed.

New in version 0.13.

ezdx.f.math.rational_spline_from_ellipse(ellipse: ConstructionEllipse, segments: int = 1) → BSpline

Returns a rational B-splines for an elliptic arc.

Parameters

• ellipse – ellipse parameters as ConstructionEllipse object
• segments – count of spline segments, at least one segment for each quarter (pi/2), 1 for as few as needed.

New in version 0.13.

ezdx.f.math.cubic_bezier_from_arc(center: Vector = (0, 0), radius:float=1, start_angle: float = 0, end_angle: float = 360, segments: int = 1) → Iterable[Bezier4P]

Returns an approximation for a circular 2D arc by multiple cubic Bézier curves.

Parameters

• center – circle center as Vector 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), 1 for as few as needed.
New in version 0.13.

```
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 spline segments, at least one segment for each quarter (pi/2), 1 for as few as needed.

New in version 0.13.

```
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

New in version 0.13.

**Transformation Classes**

**OCS Class**

```
class ezdxf.math.OCS(extrusion: Vertex = Vector(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**

```
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: \( \mathbf{ux} = (1, 0, 0), \mathbf{uy} = (0, 1, 0), \mathbf{uz} = (0, 0, 1) \).

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**

\( \mathbf{ux} \)  
\( \mathbf{ux} \) – x-axis unit vector

\( \mathbf{uy} \)  
\( \mathbf{uy} \) – y-axis unit vector

\( \mathbf{uz} \)  
\( \mathbf{uz} \) – z-axis unit vector

**is_cartesian**  
Returns True if cartesian coordinate system.

**copy()** → UCS  
Returns a copy of this UCS.  
New in version 0.11.

**to_wcs**(point: Vertex) → Vector  
Returns WCS point for UCS point.

**points_to_wcs**(points: Iterable[Vertex]) → Iterable[Vector]  
Returns iterable of WCS vectors for UCS points.

**direction_to_wcs**(vector: Vertex) → Vector  
Returns WCS direction for UCS vector without origin adjustment.

**from_wcs**(point: Vertex) → Vector  
Returns UCS point for WCS point.

**points_from_wcs**(points: Iterable[Vertex]) → Iterable[Vector]  
Returns iterable of UCS vectors from WCS points.

**direction_from_wcs**(vector: Vertex) → Vector  
Returns UCS vector for WCS vector without origin adjustment.

**to_ocs**(point: Vertex) → Vector  
Returns OCS vector for UCS point.

The **OCS** is defined by the z-axis of the **UCS**.

**points_to_ocs**(points: Iterable[Vertex]) → Iterable[Vector]  
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

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.
New in version 0.11.

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.
New in version 0.11.

Parameters
angle – rotation angle in radians

rotate_local_y (angle: float) → UCS
Returns a new rotated UCS, rotation axis is the local y-axis.
New in version 0.11.

Parameters
angle – rotation angle in radians

rotate_local_z (angle: float) → UCS
Returns a new rotated UCS, rotation axis is the local z-axis.
New in version 0.11.

Parameters
angle – rotation angle in radians

shift (delta: Vertex) → UCS
Shifts current UCS by delta vector and returns self.
New in version 0.11.

Parameters
delta – shifting vector

moveto (location: Vertex) → UCS
Place current UCS at new origin location and returns self.
New in version 0.11.

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) → 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) → 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) → 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) → 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) → 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** ezdxm.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.

__repr__() → str
Returns the representation string of the matrix: `Matrix44((col0, col1, col2, col3), (.. .), (...), (...))`

set (*args) → None
Set matrix values.
• set() creates the identity matrix.
• set(values) values is an iterable with the 16 components of the matrix.
• set(row1, row2, row3, row4) four rows, each row with four values.

get_row (row: int) → Tuple[float, ...]
Get row as list of of four float values.

Parameters
row – row index [0 .. 3]

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

get_col (col: int) → Tuple[float, ...]
Returns a column as a tuple of four floats.

Parameters
col – column index [0 .. 3]

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

copy () → Matrix44
Returns a copy of same type.

__copy__() → Matrix44
Returns a copy of same type.

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.

classmethod translate (dx: float, dy: float, dz: float) → Matrix44
Returns a translation matrix for translation vector (dx, dy, dz).

classmethod x_rotate (angle: float) → Matrix44
Returns a rotation matrix about the x-axis.

Parameters
angle – rotation angle in radians

classmethod y_rotate (angle: float) → Matrix44
Returns a rotation matrix about the y-axis.

Parameters
angle – rotation angle in radians
**classmethod z_rotate**(angle: float) → Matrix44
Returns a rotation matrix about the z-axis.

**Parameters**
- **angle** – rotation angle in radians

**classmethod axis_rotate**(axis: Vertex, angle: float) → Matrix44
Returns a rotation matrix about an arbitrary axis.

**Parameters**
- **axis** – rotation axis as (x, y, z) tuple or Vector object
- **angle** – rotation angle in radians

**classmethod xyz_rotate**(angle_x: float, angle_y: float, angle_z: float) → Matrix44
Returns a rotation matrix for rotation about each axis.

**Parameters**
- **angle_x** – rotation angle about x-axis in radians
- **angle_y** – rotation angle about y-axis in radians
- **angle_z** – rotation angle about z-axis in radians

**classmethod perspective_projection**(left: float, right: float, top: float, bottom: float, near: float, far: float) → Matrix44
Returns a matrix for a 2D projection.

**Parameters**
- **left** – Coordinate of left of screen
- **right** – Coordinate of right of screen
- **top** – Coordinate of the top of the screen
- **bottom** – Coordinate of the bottom of the screen
- **near** – Coordinate of the near clipping plane
- **far** – Coordinate of the far clipping plane

**classmethod perspective_projection_fov**(fov: float, aspect: float, near: float, far: float) → Matrix44
Returns a matrix for a 2D projection.

**Parameters**
- **fov** – The field of view (in radians)
- **aspect** – The aspect ratio of the screen (width / height)
- **near** – Coordinate of the near clipping plane
- **far** – Coordinate of the far clipping plane

**static chain**(matrices: Iterable[Matrix44]) → 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

```python
__hash__() → int
Returns hash value of matrix.

__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.

fast_mul(other: Matrix44) → Matrix44
Multiplies this matrix with other matrix.

Assumes that both matrices have a right column of (0, 0, 0, 1). This is True for matrices composed of rotations, translations and scales. fast_mul is approximately 25% quicker than the *= operator.

transform(vector: Vertex) → ezdxf.math.vector.Vector
Returns a transformed vertex.

transform_direction(vector: Vertex, normalize=False) → ezdxf.math.vector.Vector
Returns a transformed direction vector without translation.

transform_vertices(vectors: Iterable[Vertex]) → Iterable[ezdxf.math.vector.Vector]
Returns an iterable of transformed vertices.

transform_directions(vectors: Iterable[Vertex], normalize=False) → Iterable[ezdxf.math.vector.Vector]
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**

---

5.4. Reference
class ezdxf.math.Vector(*args)

This is an immutable universal 3D vector object. This class is optimized for universality not for speed. Immutable means you can’t change (x, y, z) components after initialization:

\[
\begin{align*}
v1 &= \text{Vector}(1, 2, 3) \\
v2 &= v1 \\
v2.z &= 7 \quad \# \text{this is not possible, raises AttributeError} \\
v2 &= \text{Vector}(v2.x, v2.y, 7) \quad \# \text{this creates a new Vector() object} \\
\text{assert} \ v1.z &= 3 \quad \# \text{and v1 remains unchanged}
\end{align*}
\]

Vector initialization:

- Vector(), returns Vector(0, 0, 0)
- Vector((x, y)), returns Vector(x, y, 0)
- Vector((x, y, z)), returns Vector(x, y, z)
- Vector(x, y), returns Vector(x, y, 0)
- Vector(x, y, z).returns Vector(x, y, z)

Addition, subtraction, scalar multiplication and scalar division left and right handed are supported:

\[
\begin{align*}
v &= \text{Vector}(1, 2, 3) \\
v + (1, 2, 3) &= \text{Vector}(2, 4, 6) \\
(1, 2, 3) + v &= \text{Vector}(2, 4, 6) \\
v - (1, 2, 3) &= \text{Vector}(0, 0, 0) \\
(1, 2, 3) - v &= \text{Vector}(0, 0, 0) \\
v * 3 &= \text{Vector}(3, 6, 9) \\
3 * v &= \text{Vector}(3, 6, 9) \\
\text{Vector}(3, 6, 9) / 3 &= \text{Vector}(1, 2, 3) \\
-\text{Vector}(1, 2, 3) &= (-1, -2, -3)
\end{align*}
\]

Comparison between vectors and vectors or tuples is supported:

\[
\begin{align*}
\text{Vector}(1, 2, 3) &< \text{Vector}(2, 2, 2) \\
(1, 2, 3) &< \text{tuple(\text{Vector}(2, 2, 2))} \quad \# \text{conversion necessary} \\
\text{Vector}(1, 2, 3) &= (1, 2, 3)
\end{align*}
\]

bool(Vector(1, 2, 3)) is True  
bool(Vector(0, 0, 0)) is False

\[
\begin{align*}
x &\quad \text{x-axis value} \\
y &\quad \text{y-axis value} \\
z &\quad \text{z-axis value} \\
xy &\quad \text{Vector as (x, y, 0), projected on the xy-plane.} \\
xyz &\quad \text{Vector as (x, y, z) tuple.}
\end{align*}
\]
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 Vector(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__ () \rightarrow str
Return "(x, y, z)" as string.

__repr__ () \rightarrow str
Return 'Vector(x, y, z)' as string.

__len__ () \rightarrow int
Returns always 3.

__hash__ () \rightarrow int
Returns hash value of vector, enables the usage of vector as key in set and dict.

copy () \rightarrow Vector
Returns a copy of vector as Vector object.

__copy__ () \rightarrow Vector
Returns a copy of vector as Vector object.

__deepcopy__ (memodict: dict) \rightarrow Vector
copy.deepcopy() support.

__getitem__ (index: int) \rightarrow float
Support for indexing:
• v[0] is v.x
• v[1] is v.y
• v[2] is v.z

__iter__ () \rightarrow Iterable[float]
Returns iterable of x-, y- and z-axis.

__abs__ () \rightarrow float
Returns length (magnitude) of vector.

replace (x: float = None, y: float = None, z: float = None) \rightarrow Vector
Returns a copy of vector with replaced x-, y- and/or z-axis.
classmethod generate(items: Iterable[Vertex]) → Iterable[Vector]
Returns an iterable of Vector objects.

classmethod list(items: Iterable[Vertex]) → List[Vector]
Returns a list of Vector objects.

classmethod from_angle(angle: float, length: float = 1.) → Vector
Returns a Vector object from angle in radians in the xy-plane, z-axis = 0.

classmethod from_deg_angle(angle: float, length: float = 1.) → Vector
Returns a Vector object from angle in degrees in the xy-plane, z-axis = 0.

orthogonal(ccw: bool = True) → Vector
Returns orthogonal 2D vector, z-axis is unchanged.

Parameters
ccw – counter clockwise if True else clockwise

lerp(other: Any, factor=.5) → Vector
Returns linear interpolation between self and other.

Parameters
• other – end point as Vector compatible object
• factor – interpolation factor (0 = self, 1 = other, 0.5 = mid point)

is_parallel(other: Vector, abs_tolr=1e-12) → bool
Returns True if self and other are parallel to vectors.

project(other: Any) → Vector
Returns projected vector of other onto self.

normalize(length: float = 1.) → Vector
Returns normalized vector, optional scaled by length.

reversed() → Vector
Returns negated vector (-self).

isclose(other: Any, abs_tol: float = 1e-12) → bool
Returns True if self is close to other. Uses math.isclose() to compare all axis.

__neg__() → Vector
Returns negated vector (-self).

__bool__() → bool
Returns True if vector is not (0, 0, 0).

__eq__(other: Any) → bool
Equal operator.

Parameters
other – Vector compatible object

__lt__(other: Any) → bool
Lower than operator.

Parameters
other – Vector compatible object

__add__(other: Any) → Vector
Add operator: self + other

Parameters
other – Vector compatible object

__radd__(other: Any) → Vector
RAdd operator: other + self

Parameters
other – Vector compatible object
__sub__ (other: Any) → Vector
Sub operator: self - other
Parameters other = Vector compatible object

__rsub__ (other: Any) → Vector
RSub operator: other - self
Parameters other = Vector compatible object

__mul__ (other: float) → Vector
Mul operator: self * other
Parameters other = scale factor

__rmul__ (other: float) → Vector
RMul operator: other * self
Parameters other = scale factor

__truediv__ (other: float) → Vector
Div operator: self / other
Parameters other = scale factor

__div__ (other: float) → Vector
Div operator: self / other
Parameters other = scale factor

__rtruediv__ (other: float) → Vector
RDiv operator: other / self
Parameters other = scale factor

__rdiv__ (other: float) → Vector
RDiv operator: other / self
Parameters other = scale factor

dot (other: Any) → float
Dot operator: self . other
Parameters other = Vector compatible object

cross (other: Any) → Vector
Dot operator: self x other
Parameters other = Vector compatible object

distance (other: Any) → float
Returns distance between self and other vector.

angle_about (base: Vector, target: Vector) → 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: Any) → float
Returns angle between self and other in radians. +angle is counter clockwise orientation.
Parameters other = Vector compatible object
rotate \( (\text{angle}: \text{float}) \rightarrow \text{Vector} \)

Returns vector rotated about \( \text{angle} \) around the z-axis.

- **Parameters**
  - angle: \( \text{angle} \) in radians

rotate_deg \( (\text{angle}: \text{float}) \rightarrow \text{Vector} \)

Returns vector rotated about \( \text{angle} \) around the z-axis.

- **Parameters**
  - angle: \( \text{angle} \) in degrees

```python
ezdxf.math.X_AXIS
Vector(1, 0, 0)

ezdxf.math.Y_AXIS
Vector(0, 1, 0)

ezdxf.math.Z_AXIS
Vector(0, 0, 1)

ezdxf.math.NULLVEC
Vector(0, 0, 0)
```

**Vec2**

```python
class ezdxf.math.Vec2(v)
```

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 Vector 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 Vector objects.

- **Parameters**
  - \(v\) – vector object with \(x\) and \(y\) attributes/properties or a sequence of float \([x, y, \ldots]\) or \(x\)-axis as float if argument \(y\) is not \(None\)
  - \(y\) – second float for Vec2 \((x, y)\)

Vec2 implements a subset of Vector.

**Plane**

```python
class ezdxf.math.Plane(normal: Vector, distance: float)
```

Represents a plane in 3D space as normal vector and the perpendicular distance from origin.

New in version 0.11.

- **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**(\(a: \text{Vector}, b: \text{Vector}, c: \text{Vector}\)) \rightarrow \text{Plane}

Returns a new plane from 3 points in space.

**classmethod from_vector**(\(\text{vector}\)) \rightarrow \text{Plane}

Returns a new plane from a location vector.
copy ( ) → Plane
Returns a copy of the plane.

signed_distance_to (v: Vector) → 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: Vector) → float
Returns absolute (unsigned) distance of vertex v to plane.

is_coplanar_vertex (v: Vector, 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 Vector objects

extmin
“lower left” corner of bounding box

extmax
“upper right” corner of bounding box

center
Returns center of bounding box.

extend (vertices: Iterable[Vertex]) → None
Extend bounds by vertices.

Parameters vertices – iterable of (x, y, z) tuples or Vector objects

has_data
Returns True if data is available

inside (vertex: Vertex) → bool
Returns True if vertex is inside bounding box.

size
Returns size of bounding box.

BoundingBox2d
class ezdxf.math.BoundingBox2d (vertices: Iterable[Vertex] = None)
Optimized 2D bounding box.

Parameters vertices – iterable of (x, y [, z]) tuples or Vector objects

extmin
“lower left” corner of bounding box

extmax
“upper right” corner of bounding box
center
Returns center of bounding box.

extend(vertices: Iterable[Vertex]) → None
Extend bounds by vertices.

Parameters vertices – iterable of (x, y[, z]) tuples or Vector objects

has_data
Returns True if data is available

inside(vertex: Vertex) → bool
Returns True if vertex is inside bounding box.

size
Returns size of 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.

**__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 two 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 (e.g. test for circle touch point)

**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 <strong>other</strong> circle at one point</td>
</tr>
<tr>
<td>2</td>
<td>circle intersects with the <strong>other</strong> 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 \texttt{Vec2}

radius
radius as float

start_angle
start angle in degrees

end_angle
end angle in degrees

start_angle_rad
start angle in radians.

end_angle_rad
end angle in radians.

start_point
start point of arc as \texttt{Vec2}.

end_point
end point of arc as \texttt{Vec2}.

bounding_box
bounding box of arc as \texttt{BoundingBox2d}.

angles (\textit{num: int}) \to \textit{Iterable[float]}
Returns \textit{num} angles from start- to end angle in degrees in counter clockwise order.
All angles are normalized in the range from [0, 360).

vertices (\textit{a: Iterable[float]}) \to \textit{Iterable[ezdxf.math.vector.Vec2]}
Yields vertices on arc for angles in iterable \textit{a} in WCS as location vectors.

Parameters
\textbf{a} – angles in the range from 0 to 360 in degrees, arc goes counter clockwise around the z-axis, WCS x-axis = 0 deg.

vertices (\textit{a: Iterable[float]}) \to \textit{Iterable[ezdxf.math.vector.Vec2]}
Yields tangents on arc for angles in iterable \textit{a} in WCS as direction vectors.

Parameters
\textbf{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 (\textit{a: Iterable[float]}) \to \textit{Iterable[ezdxf.math.vector.Vec2]}
Yields tangents on arc for angles in iterable \textit{a} in WCS as direction vectors.

Parameters
\textbf{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 (\textit{dx: float, dy: float}) \to \textit{ConstructionArc}
Move arc about \textit{dx} in x-axis and about \textit{dy} in y-axis, returns \textit{self} (floating interface).

Parameters

- \textbf{dx} – translation in x-axis
- \textbf{dy} – translation in y-axis

scale_uniform (\textit{s: float}) \to \textit{ConstructionArc}
Scale arc inplace uniform about \textit{s} in x- and y-axis, returns \textit{self} (floating interface).

translate (\textit{dx: float, dy: float}) \to \textit{ConstructionArc}
Move arc about \textit{dx} in x-axis and about \textit{dy} in y-axis, returns \textit{self} (floating interface).

Parameters

- \textbf{dx} – translation in x-axis
- \textbf{dy} – translation in y-axis

rotate_z (\textit{angle: float}) \to \textit{ConstructionArc}
Rotate arc inplace about z-axis, returns \textit{self} (floating interface).

Parameters

- \textbf{angle} – rotation angle in degrees

classmethod from_2p_angle (\textit{start_point: Vertex, end_point: Vertex, angle: float, ccw: bool = True}) \to \textit{ConstructionArc}
Create arc from two points and enclosing angle. Additional precondition: arc goes by default in counter clockwise orientation from \textit{start_point} to \textit{end_point}, can be changed by \textit{ccw} = False.
Parameters

- **start_point** – start point as \texttt{Vec2} compatible object
- **end_point** – end point as \texttt{Vec2} compatible object
- **angle** – enclosing angle in degrees
- **ccw** – counter clockwise direction if True

**classmethod from\_2p\_radius** (*start_point*: \texttt{Vertex}, *end_point*: \texttt{Vertex}, *radius*: float, *ccw*: bool = True, *center\_is\_left*: bool = True) → \texttt{ConstructionArc}

Create arc from two points and arc radius. Additional precondition: arc goes by default in counter clockwise orientation from \texttt{start_point} to \texttt{end_point} can be changed by \texttt{ccw} = \texttt{False}.

The parameter \texttt{center\_is\_left} defines if the center of the arc is left or right of the line from \texttt{start\_point} to \texttt{end\_point}. Parameter \texttt{ccw} = \texttt{False} swaps start- and end point, which inverts the meaning of \texttt{center\_is\_left}.

Parameters

- **start_point** – start point as \texttt{Vec2} compatible object
- **end_point** – end point as \texttt{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*: \texttt{Vertex}, *end_point*: \texttt{Vertex}, *def\_point*: \texttt{Vertex}, *ccw*: bool = True) → \texttt{ConstructionArc}

Create arc from three points. Additional precondition: arc goes in counter clockwise orientation from \texttt{start\_point} to \texttt{end\_point}.

Parameters

- **start_point** – start point as \texttt{Vec2} compatible object
- **end_point** – end point as \texttt{Vec2} compatible object
- **def\_point** – additional definition point as \texttt{Vec2} compatible object
- **ccw** – counter clockwise direction if True

**add\_to\_layout** (*layout*: \texttt{BaseLayout}, *ucs*: \texttt{UCS} = None, *dxfattribs*: dict = None) → \texttt{Arc}

Add arc as DXF \texttt{Arc} entity to a layout.

Supports 3D arcs by using an \texttt{UCS}. An \texttt{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 \texttt{BaseLayout} object
- **ucs** – place arc in 3D space by \texttt{UCS} object
- **dxfattribs** – additional DXF attributes for DXF \texttt{Arc} entity
ConstructionEllipse

class ezdxm.math.ConstructionEllipse(center: Vertex = Vector(0.0, 0.0, 0.0), major_axis: Vertex = Vector(1.0, 0.0, 0.0), extrusion: Vertex = Vector(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 Vector

**major_axis**
major axis as Vector

**minor_axis**
minor axis as Vector, automatically calculated from major_axis and extrusion.

**extrusion**
extrusion vector (normal of ellipse plane) as Vector

**ratio**
ratio of minor axis to major axis (float)

**start**
start param in radians (float)

**end**
end param in radians (float)

**start_point**
Returns start point of ellipse as Vector.

**end_point**
Returns end point of ellipse as Vector.

**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, 2pi).

**vertices**(params: Iterable[float]) → Iterable[ezdxm.math.vector.Vector]
Yields vertices on ellipse for iterable params in WCS.
Parameters **params** – param values in the range from 0 to 2*π in radians, param goes counter clockwise around the extrusion vector, major_axis = local x-axis = 0 rad.

**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.Vector]

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 e兹dxf.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

center
    box center

width
    box width

height
    box height

angle
    rotation angle in degrees

corners
    box corners as sequence of Vec2 objects.

bounding_box
    BoundingBox2d

incircle_radius
    incircle radius

circumcircle_radius
    circum circle radius

__iter__() → Iterable[Vec2]
    Iterable of box corners as Vec2 objects.

__getitem__(corner) → Vec2
    Get corner by index corner, list like slicing is supported.

__repr__() → str
    Returns string representation of box as ConstructionBox(center, width, height, angle)

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: float, dy: float) → 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: float, dh: float) → None*
Expand box: *dw* expand width, *dh* expand height.

**scale** *(sw: float, sh: float) → None*
Scale box: *sw* scales width, *sh* scales height.

**rotate** *(angle: float) → None*
Rotate box by *angle* in degrees.

**is_inside** *(point: Vertex) → bool*
Returns True if *point* is inside of box.

**is_any_corner_inside** *(other: ConstructionBox) → bool*
Returns True if any corner of *other* box is inside this box.

**is_overlapping** *(other: ConstructionBox) → bool*
Returns True if this box and *other* box do overlap.

**border_lines** *(*) → Sequence[ConstructionLine]*
Returns border lines of box as sequence of *ConstructionLine*.

**intersect** *(line: ConstructionLine) → List[Vec2]*
Returns 0, 1 or 2 intersection points between *line* and box border lines.

**Parameters**

- **line** – line to intersect with border lines

**Returns**

- list of intersection points

<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 *(vertices: Iterable[Vertex] = None)*

2D geometry object as list of *Vec2* objects, vertices can be moved, rotated and scaled.

**Parameters**

- **vertices** – iterable of *Vec2* compatible objects.

**vertices**

List of *Vec2* objects

**bounding_box**

*BoundingBox2d*

**__len__** *(*) → int*

Returns count of vertices.

**__getitem__** *(item) → Vec2*

Get vertex by index *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.

   New in version 0.11.

   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, using an uniform open knot vector ("clamped").

   Parameters
   • control_points – iterable of control points as Vector compatible objects
   • order – spline order (degree + 1)
   • knots – iterable of knot values
   • weights – iterable of weight values

count
   Control points as list of Vector

count
   Count of control points, (n + 1 in text book notation).

degree
   Degree (p) of B-spline = order - 1
**order**
Order of B-spline = degree + 1

**max_t**
Biggest knot value.

**is_rational**
Returns True if curve is a rational B-spline. (has weights)

**knots()** → List[float]
Returns a list of knot values as floats, the knot vector always has order + count values (n + p + 2 in text book notation).

**normalize_knots()**
Normalize knot vector into range [0, 1].

**weights()** → List[float]
Returns a list of weights values as floats, one for each control point or an empty list.

**params(segments: int)** → Iterable[float]
Yield evenly spaced parameters from 0 to max_t for given segment count.

**reverse()** → BSpline
Returns a new BSpline with revered control point order.

**point(t: float)** → Vector
Returns point for parameter t.

**Parameters**
- t – parameter in range [0, max_t]

**points(t: float)** → List[Vector]
Yields points for parameter vector t.

**Parameters**
- t – parameters in range [0, max_t]

**derivative(t: float, n: int=2)** → List[Vector]
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 Vector objects

**derivatives(t: Iterable[float], n: int=2)** → Iterable[List[Vector]]
Yields points and derivatives up to n <= 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 <= degree

**Returns**
List of n+1 values as Vector objects

**insert_knot(t: float)** → None
Insert additional knot, without altering the curve shape.

**Parameters**
- t – position of new knot 0 < t < max_t
approximate (segments: int = 20) → Iterable[Vector]
Approximates curve by vertices as Vector objects, vertices count = segments + 1.

static from_ellipse (ellipse: ConstructionEllipse) → BSpline
Returns the ellipse as BSpline of 2nd degree with as few control points as possible.

static from_arc (arc: ConstructionArc) → BSpline
Returns the arc as BSpline of 2nd degree with as few control points as possible.

static from_fit_points (points: Iterable[Vertex], degree:int=3, method='chord') → BSpline
Returns BSpline defined by fit points.

static ellipse_approximation (ellipse: ConstructionEllipse, num:int=16) → BSpline
Returns an ellipse approximation as BSpline with num control points.

static arc_approximation (arc: ConstructionArc, num:int=16) → BSpline
Returns an arc approximation as BSpline with num control points.

transform (m: Matrix44) → BSpline
Transform B-spline by transformation matrix m inplace.

New in version 0.13.

bezier_decomposition () → Iterable[List[Vector]]
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. B-spline of 3rd degree yields cubic Bézier curves of 4 control points.

cubic_bezier_approximation (level: int = 3, segments: int = None) → Iterable[Bezier4P]
Approximate arbitrary B-splines (degree != 3 and/or rational) by multiple segments of cubic Bézier curves. The choice of cubic Bézier curves is based on the widely support of this curves by many render backends. For cubic non-rational B-splines, which is maybe the most common used B-spline, is bezier_decomposition() the better choice.

1. approximation by level: an educated guess for 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 >= 3 is recommended.

2. approximation by a given count of evenly distributed approximation segments.

Parameters

• level – subdivision level of approximation segments (ignored if argument segments != None)
• segments – absolute count of approximation segments

Returns Yields control points of cubic Bézier curves as Bezier4P objects

BSplineU

class ezdxf.math.BSplineU (control_points: Iterable[Vertex], order: int = 4, knots: Iterable[float] = None, weights: Iterable[float] = None)
Representation of an uniform (periodic) B-spline curve (open curve).
BSplineClosed

class ezdxf.math.BSplineClosed(control_points: Iterable[Vertex], order: int = 4, knots: Iterable[float] = None, weights: Iterable[float] = None)

Representation of a closed uniform B-spline curve (closed curve).

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 general 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.

Parameters defpoints – iterable of definition points as Vector compatible objects.

control_points

Control points as list of Vector objects.

params (segments: int) → Iterable[float]

Yield evenly spaced parameters from 0 to 1 for given segment count.

approximate (segments: int = 20) → Iterable[Vector]

Approximates curve by vertices as Vector objects, vertices count = segments + 1.

point (t: float) → Vector

Returns a point for parameter t in range [0, 1] as Vector object.

points (t: Iterable[float]) → Iterable[Vector]

Yields multiple points for parameters in vector t as Vector objects. Parameters have to be in range [0, 1].

derivative (t: float) → Tuple[Vector, Vector, Vector]

Returns (point, 1st derivative, 2nd derivative) tuple for parameter t in range [0, 1] as Vector objects.

derivatives (t: Iterable[float]) → Iterable[Tuple[Vector, Vector, Vector]]

Returns multiple (point, 1st derivative, 2nd derivative) tuples for parameter vector t as Vector objects. Parameters in range [0, 1]

Bezier4P

class ezdxf.math.Bezier4P (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 Vec2 objects
• 3D control points in, returns 3D results as Vector objects

Parameters defpoints – iterable of definition points as Vec2 or Vector compatible objects.

control_points

control points as list of (x, y, z), z-axis is 0 for 2D curves.
to2d() → Bezier4P
Returns the bezier curve with 2d control points. (discards the z-axis)

to3d() → Bezier4P
Returns the bezier curve with 3d control points.

point (t: float) → Union[Vector, Vec2]
Returns point for location t' at the Bézier curve.
Parameters t – curve position in the range [0, 1]

tangent (t: float) → Union[Vector, Vec2]
Returns direction vector of tangent for location t at the Bézier curve.
Parameters t – curve position in the range [0, 1]

approximate (segments: int) → Iterable[Union[Vector, Vec2]]
Approximate Bézier curve by vertices, yields segments + 1 vertices as \((x, y, [, z])\) tuples.
Parameters segments – count of segments for approximation

approximated_length (segments: int = 100) → float
Returns estimated length of Bézier curve as approximation by line segments.

BezierSurface

class ezdxf.math.BezierSurface (defpoints: List[List[Sequence[float]]])
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 defpoints – matrix (list of lists) of \(m\) rows and \(n\) columns: \([m1n1, m1n2, ...],

[m2n1, m2n2, ...] \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: float, v: float) → Sequence[float]
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) → List[List[Sequence[float]]]
Approximate surface as grid of \((x, y, z)\) tuples.

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 \((x, y, z)\) tuples.

EulerSpiral

class ezdxf.math.EulerSpiral (curvature: float = 1.0)
This class represents an euler spiral (clothoid) for curvature (Radius of curvature).
This is a parametric curve, which always starts at the origin = \((0, 0)\).
Parameters

**curvature** – radius of curvature

**radius** \((t: \text{float}) \rightarrow \text{float}\)
Get radius of circle at distance \(t\).

**tangent** \((t: \text{float}) \rightarrow \text{Vector}\)
Get tangent at distance \(t\) as \(\text{Vector}\) object.

**distance** \((\text{radius: float}) \rightarrow \text{float}\)
Get distance \(L\) from origin for \(\text{radius}\).

**point** \((t: \text{float}) \rightarrow \text{Vector}\)
Get point at distance \(t\) as \(\text{Vector}\) object.

**circle_center** \((t: \text{float}) \rightarrow \text{Vector}\)
Get circle center at distance \(t\).

Changed in version 0.10: renamed from **circle_midpoint**

**approximate** \((\text{length: float, segments: int}) \rightarrow \text{Iterable[Vector]}\)
Approximate curve of length with line segments.
Generates \(\text{segments}+1\) vertices as \(\text{Vector}\) 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 \(n\times n\) Matrix \(A\ . x = B\), right-hand side quantities as \(n\times m\) 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

New in version 0.13.
ezdxf.math.gauss_jordan_inverse \( (A: \text{Iterable}[\text{Iterable}[\text{float}]] \) → \text{Matrix} \\
Returns the inverse of matrix \( A \) as \text{Matrix} object.

**Hint:** For small matrices \((n<10)\) is this function faster than \text{LUDecomposition}(m).inverse() and as fast even if the decomposition is already done.

**Raises** \text{ZeroDivisionError} – singular matrix

New in version 0.13.

ezdxf.math.gauss_vector_solver \( (A: \text{Iterable}[\text{Iterable}[\text{float}]], B: \text{Iterable}[\text{float}] \) → \text{List}[\text{float}] \\
Solves the linear equation system given by a \text{n}x\text{n} Matrix \( A \cdot x = B \), right-hand side quantities as vector \( B \) with \text{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 \([a11, a12, \ldots, a1n], [a21, a22, \ldots, a2n], [a21, a22, \ldots, a2n], \ldots, [an1, an2, \ldots, ann]\)
- \( B \) – vector \([b1, b2, \ldots, bn]\)

**Returns** vector as list of floats

**Raises** \text{ZeroDivisionError} – singular matrix

New in version 0.13.

ezdxf.math.gauss_matrix_solver \( (A: \text{Iterable}[\text{Iterable}[\text{float}]], B: \text{Iterable}[\text{Iterable}[\text{float}]]) \) → \text{Matrix} \\
Solves the linear equation system given by a \text{n}x\text{n} Matrix \( A \cdot x = B \), right-hand side quantities as \text{n}x\text{m} Matrix \( B \) by the Gauss-Elimination algorithm, which is faster than the Gauss-Jordan algorithm.

Reference implementation for error checking.

**Parameters**

- \( A \) – matrix \([a11, a12, \ldots, a1n], [a21, a22, \ldots, a2n], [a21, a22, \ldots, a2n], \ldots, [an1, an2, \ldots, ann]\)
- \( B \) – matrix \([b11, b12, \ldots, b1m], [b21, b22, \ldots, b2m], \ldots, [bn1, bn2, \ldots, bnm]\)

**Returns** matrix as \text{Matrix} object

**Raises** \text{ZeroDivisionError} – singular matrix

New in version 0.13.

ezdxf.math.tridiagonal_vector_solver \( (A: \text{Iterable}[\text{Iterable}[\text{float}]], B: \text{Iterable}[\text{float}]) \) → \text{List}[\text{float}] \\
Solves the linear equation system given by a tri-diagonal \text{n}x\text{n} Matrix \( A \cdot x = B \), 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 \text{ZeroDivisionError} exception occurs, the equation system can possibly be solved by \text{BandedMatrixLU}(A, 1, 1).solve_vector(B)

**Parameters**

- \( A \) – diagonal matrix \([a0..an-1], [b0..bn-1], [c0..cn-1]]\)
• \( \mathbf{B} \) – iterable of floats \([b_0, b_1, \ldots, b_n]\)

Returns  list of floats  
 Raises  ZeroDivisionError – singular matrix

New in version 0.13.

```python
ezdxf.math.tridiagonal_matrix_solver(A: Iterable[Iterable[float]], B: Iterable[Iterable[float]]) → Matrix
```

Solves the linear equation system given by a tri-diagonal nxn Matrix \( \mathbf{A} \) \( \mathbf{x} = \mathbf{B} \), right-hand side quantities as nxm Matrix \( \mathbf{B} \). Matrix \( \mathbf{A} \) is diagonal matrix defined by 3 diagonals \([-1 (a), 0 (b), +1 (c)]\).

Note: \( a_0 \) is not used but has to be present, \( c_{n-1} \) is also not used and must not be present.

If an ZeroDivisionError exception occurs, the equation system can possibly be solved by BandedMatrixLU(\( \mathbf{A}, 1, 1 \)).solve_vector(\( \mathbf{B} \))

Parameters

• \( \mathbf{A} \) – diagonal matrix \([a_0..a_{n-1}], [b_0..b_{n-1}], [c_0..c_{n-1}]\]

• \( \mathbf{B} \) – 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  
 Raises  ZeroDivisionError – singular matrix

New in version 0.13.

```python
ezdxf.math.banded_matrix(A: Matrix, check_all=True) → Tuple[int, int]
```

Transform matrix \( \mathbf{A} \) into a compact banded matrix representation. Returns compact representation as Matrix object and lower- and upper band count \( m_1 \) and \( m_2 \).

Parameters

• \( \mathbf{A} \) – input Matrix

• \( \text{check\_all} \) – check all diagonals if True or abort testing after first all zero diagonal if False.

```python
ezdxf.math.detect_banded_matrix(A: Matrix, check_all=True) → Tuple[int, int]
```

Returns lower- and upper band count \( m_1 \) and \( m_2 \).

Parameters

• \( \mathbf{A} \) – input Matrix

• \( \text{check\_all} \) – check all diagonals if True or abort testing after first all zero diagonal if False.

```python
ezdxf.math.compact_banded_matrix(A: Matrix, m1: int, m2: int) → Matrix
```

Returns compact banded matrix representation as Matrix object.

Parameters
• A – matrix to transform
• m1 – lower band count, excluding main matrix diagonal
• m2 – upper band count, excluding main matrix diagonal

\[
\text{ezdxf.math.freeze_matrix}(A: \text{Union[MatrixData, Matrix]}) \rightarrow \text{Matrix}
\]

Returns a frozen matrix, all data is stored in immutable tuples.

Matrix Class

class ezdxf.math.Matrix(items: Any = None, shape: Tuple[int, int] = None, matrix: List[List[float]] = 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 \text{Matrix.matrix}.

The matrix can be frozen by function \text{freeze_matrix()} or method \text{Matrix.freeze()}, than the data is stored in immutable tuples.

Initialization:

• Matrix(shape=(rows, cols)) . . . new matrix filled with zeros
• Matrix(matrix[, shape=(rows, cols)]) . . . from copy of matrix and optional reshape
• Matrix([[row_0], [row_1], . . . , [row_n]]) . . . from \text{Iterable[Iterable[float]]}
• Matrix([a1, a2, . . . , an], shape=(rows, cols)) . . . from \text{Iterable[float]} and shape

New in version 0.13.

nrows
Count of matrix rows.

cols
Count of matrix columns.

shape
Shape of matrix as (n, m) tuple for n rows and m columns.

static reshape(items: \text{Iterable[float]}, shape: \text{Tuple[int, int]}) \rightarrow \text{ezdxf.math.linalg.Matrix}

Returns a new matrix for iterable \text{items} in the configuration of \text{shape}.

classmethod identity(shape: \text{Tuple[int, int]}) \rightarrow \text{ezdxf.math.linalg.Matrix}

Returns the identity matrix for configuration \text{shape}.

row(index) \rightarrow \text{List[float]}
Returns row \text{index} as list of floats.

iter_row(index) \rightarrow \text{Iterable[float]}
Yield values of row \text{index}.

col(index) \rightarrow \text{List[float]}
Return column \text{index} as list of floats.

iter_col(index) \rightarrow \text{Iterable[float]}
Yield values of column \text{index}.

diag(index) \rightarrow \text{List[float]}
Returns diagonal \text{index} as list of floats.

An \text{index} of 0 specifies the main diagonal, negative values specifies diagonals below the main diagonal and positive values specifies diagonals above the main diagonal.
iter_diag(index) \to \text{Iterable[float]}
Yield values of diagonal index, see also \text{diag()}.

rows() \to \text{List[List[float]]}
Return a list of all rows.

cols() \to \text{List[List[float]]}
Return a list of all columns.

set_row(index: int, items: \text{Union[float, Iterable[float]]} = 1.0) \to \text{None}
Set row values to a fixed value or from an iterable of floats.

set_col(index: int, items: \text{Union[float, Iterable[float]]} = 1.0) \to \text{None}
Set column values to a fixed value or from an iterable of floats.

set_diag(index: int = 0, items: \text{Union[float, Iterable[float]]} = 1.0) \to \text{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.

append_row(items: \text{Sequence[float]}) \to \text{None}
Append a row to the matrix.

append_col(items: \text{Sequence[float]}) \to \text{None}
Append a column to the matrix.

swap_rows(a: int, b: int) \to \text{None}
Swap rows a and b inplace.

swap_cols(a: int, b: int) \to \text{None}
Swap columns a and b inplace.

transpose() \to \text{Matrix}
Returns a new transposed matrix.

inverse() \to \text{Matrix}
Returns inverse of matrix as new object.

determinant() \to \text{float}
Returns determinant of matrix, raises \text{ZeroDivisionError} if matrix is singular.

freeze() \to \text{Matrix}
Returns a frozen matrix, all data is stored in immutable tuples.

lu_decomp() \to \text{LUDecomposition}
Returns the LU decomposition as \text{LUDecomposition} object, a faster linear equation solver.

_getitem_(item: \text{Tuple[int, int]}) \to \text{float}
Get value by (row, col) index tuple, fancy slicing as known from numpy is not supported.

_setitem_(item: \text{Tuple[int, int]}, value: float)
Set value by (row, col) index tuple, fancy slicing as known from numpy is not supported.

eq(other: \text{Matrix}) \to \text{bool}
Returns True if matrices are equal, tolerance value for comparison is adjustable by the attribute \text{Matrix.abs_tol}.

add(other: \text{Union[Matrix, float]}) \to \text{Matrix}
Matrix addition by another matrix or a float, returns a new matrix.
**ezdxf Documentation, Release 0.13**

__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

New in version 0.13.

**nrows**  
Count of matrix rows (and cols).

**solve_vector** (B: Iterable[float]) → List[float]  
Solves the linear equation system given by the nxn Matrix A . x = B, right-hand side quantities as vector B with n elements.

**Parameters**  
B – vector [b1, b2, . . . , bn]

**Returns**  
vector as list of floats

**solve_matrix** (B: Iterable[Iterable[float]]) → Matrix  
Solves the linear equation system given by the nxn Matrix A . x = B, right-hand side quantities as nxm Matrix B.

**Parameters**  
B – matrix [[b11, b12, . . . , b1m], [b21, b22, . . . , b2m], . . . [bn1, bn2, . . . , bnm]]

**Returns**  
matrix as Matrix object

**inverse** () → Matrix  
Returns the inverse of matrix as Matrix object, raise ZeroDivisionError for a singular matrix.

**determinant** () → float  
Returns the determinant of matrix, raises ZeroDivisionError if matrix is singular.

**BandedMatrixLU Class**

class ezdxf.math.BandedMatrixLU (A: ezdxf.math.linalg.Matrix, m1: int, m2: 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 \((B: \text{Iterable}[float]) \rightarrow \text{List}[float]\)
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 \((B: \text{Iterable}[\text{Iterable}[float]]) \rightarrow \text{Matrix}\)
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 \(\text{Matrix}\) object

determinant () \(\rightarrow \text{float}\)
Returns the determinant of matrix.

## 5.4.4 Miscellaneous

### Global Options

Global options stored in \texttt{ezdxf.options} 

- \texttt{ezdxf.options.default_text_style}
  Default text styles, default value is OpenSans.

- \texttt{ezdxf.options.default_dimension_text_style}
  Default text style for Dimensions, default value is OpenSansCondensed-Light.

- \texttt{ezdxf.options.check_entity_tag_structures}
  Check app data (Application-Defined Codes) and XDATA (Extended Data) tag structures, set this option to False for a little performance boost, default value is True.

- \texttt{ezdxf.options.filter_invalid_xdata_group_codes}
  Check for invalid XDATA group codes, default value is False

- \texttt{ezdxf.options.log_unprocessed_tags}
  Log unprocessed DXF tags for debugging, default value is True

- \texttt{ezdxf.options.load_proxy_graphics}
  Load proxy graphics if True, default is False.

- \texttt{ezdxf.options.store_proxy_graphics}
  Export proxy graphics if True, default is False.

- \texttt{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.
ezdxf.options.preserve_proxy_graphics()
    Enable proxy graphic load/store support.

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

Some handy tool functions used internally by ezdxf.

ezdxf.tools.juliandate(date: datetime.datetime) → float
    ezdxf.tools.calendardate(juliandate: float) → datetime.datetime
    ezdxf.tools.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% transparency.

    ezdxf.tools.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

    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

    ezdxf.tools.guid() → str
        Returns a general unique ID, based on uuid.uuid1().

    ezdxf.tools.bytes_to_hexstr(data: bytes) → str
        Returns data bytes as plain hex string.
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

ezdxf.tools.int2rgb(value: int) → Tuple[int, int, int]

Split RGB integer value into (r, g, b) tuple.

ezdxf.tools.rgb2int(rgb: Tuple[int, int, int]) → int

Combined integer value from (r, g, b) tuple.

ezdxf.tools.aci2rgb(index: int) → Tuple[int, int, int]

Convert AutoCAD Color Index (ACI) into (r, g, b) tuple, based on default AutoCAD colors.

ezdxf.tools.normalize_text_angle(angle: float, fix_upside_down=True) → 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”

ezdxf.tools.crypt.encode(text_lines: Iterable[str]) → Iterable[str]

Encode the Standard ACIS Text (SAT) format by AutoCAD “encryption” algorithm.

ezdxf.tools.crypt.decode(text_lines: Iterable[str]) → Iterable[str]

Decode the Standard ACIS Text (SAT) format “encrypted” by AutoCAD.

5.5 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.

5.5.1 General Document

General preconditions:

```python
import ezdxf
doc = ezdxf.readfile("your_dxf_file.dxf")
msp = doc.modelspace()
```

Set/Get Header Variables

ezdxf has an interface to get and set HEADER variables:
doc.header["VarName"] = value  
value = doc.header["VarName"]

See also:
HeaderSection and online documentation from Autodesk for available header variables.

Set DXF Drawing Units

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

Sets drawing units:
$MEASUREMENT controls whether the current drawing uses imperial or metric hatch pattern and linetype files:

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

<table>
<thead>
<tr>
<th>0</th>
<th>English</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Metric</td>
</tr>
</tbody>
</table>

$LUNITS sets the linear units format for creating objects:

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

| 1 | Scientific |
| 2 | Decimal (default) |
| 3 | Engineering |
| 4 | Architectural |
| 5 | Fractional |

$AUNITS set units format for angles:

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

| 0 | Decimal degrees |
| 1 | Degrees/minutes/seconds |
| 2 | Grad |
| 3 | Radians |

$INSUNITS set default drawing units for AutoCAD DesignCenter blocks:

```
doc.header['$INSUNITS'] = 6
```
## 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 produces a HTML file `your_dxf_file.html` with a nicer layout than a plain DXF file and DXF handles as links between DXF entities, this simplifies the navigation between the DXF entities.

Changed in version 0.8.3: Since ezdxf v0.8.3, a script called `dxfpp` will be added to your Python script path:

```bash

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.
Set Initial View/Zoom for the Modelsce

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:

```
doc.set_modelsce_vport(height=10, center=(10, 10))
```

New in version 0.11.

5.5.2 DXF Viewer

A360 Viewer Problems

AutoDesk web service A360 seems to be more picky than the AutoCAD desktop applications, maybe 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:

And here in the Autodesk Online Viewer:
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:

```python
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**

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 0.11.

### 5.5.3 DXF Content

General preconditions:

```python
import ezdxf

doc = ezdxf.readfile("your_dxf_file.dxf")
msp = doc.modelspace()
```
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:

```python
doc.appids.new('YOUR_APPID')  # IMPORTANT: create an APP ID entry
circle = msp.add_circle((10, 10), 100)
circle.set_xdata('YOUR_APPID',
    [  
        (1000, 'your_web_link.org'),
        (1002, '('),
        (1000, 'some text'),
        (1002, ')'),
        (1071, 1),
        (1002, ')'),
        (1002, ')')
    ])
```

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
dimtol = dimstyle_override['dimtol']
if dimtol:
```

(continues on next page)
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 this 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.

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()

5.6 FAQ

5.6.1 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.
5.7 Rendering

The `ezdxf.render` subpackage provides helpful utilities to create complex forms, but `ezdxf` is still not a rendering engine in the sense of true graphical rendering for screen or paper.

- 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

5.7.1 Spline

Render a B-spline as 2D/3D `Polyline`, can be used with DXF R12. The advantage over `R12Spline` is the real 3D support which means the B-spline curve vertices has not to be in a plane and no hassle with `UCS` for 3D placing.

```python
class ezdxf.render.Spline

    __init__(points: Iterable[Vertex] = None, segments: int = 100)

    Parameters
    - points – spline definition points as `Vector` or `(x, y, z)` tuple
    - 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.

    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 BSpline 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 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_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 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*
5.7.2 R12Spline

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).

The result is not better than Spline, it is also just a POLYLINE entity, but as with all tools, you never know if someone needs it some day.

class ezdxf.render.R12Spline

    __init__(control_points: Iterable[Vertex], degree: int = 2, closed: bool = True)

Parameters

- control_points – B-spline control frame vertices as (x, y) tuples or Vector objects
- degree – degree of B-spline, 2 or 3 are valid values
- 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 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 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 maybe 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 Vector objects

5.7.3 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.

class ezdxf.render.Bezier
**start** *(point: Vertex, tangent: Vertex) → None*
Set start point and start tangent.

**Parameters**
- `point` – start point as Vector or (x, y, z) tuple
- `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 as Vector or (x, y, z) tuple
- `tangent1` – first control tangent as vector “left” of control point
- `tangent2` – second control tangent as vector “right” of control point, if omitted `tangent2 = -tangent1`
- `segments` – count of line segments for polyline approximation, count of line segments from previous control point to appended control point.

**render** *(layout: BaseLayout, force3d: bool = False, dxfattribs: dict = None) → None*
Render bezier curve as 2D/3D Polyline.

**Parameters**
- `layout` – BaseLayout object
- `force3d` – force 3D polyline rendering
- `dxfattribs` – DXF attributes for Polyline

### 5.7.4 EulerSpiral

Render an euler spiral as 3D Polyline or Spline.

This is a parametric curve, which always starts at the origin (0, 0).

**class** ezdxf.render.EulerSpiral

**__init__**(curvature: float = 1)

**Parameters**
- `curvature` – Radius of curvature

**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
**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

### 5.7.5 Random Paths

Random path generators for testing purpose.

```python
ezdxr.render.random_2d_path(steps=100, max_step_size=1, max_heading=pi/2, retarget=20) → Iterable[Vec2]
```

Returns a random 2D path as iterable of `Vec2` objects.

**Parameters**

- **steps** – count of vertices to generate
- **max_step_size** – max step size
- **max_heading** – limit heading angle change per step to ± max_heading/2 in radians
- **retarget** – specifies steps before changing global walking target

```python
ezdxr.render.random_3d_path(steps=100, max_step_size=1, max_heading=pi/2, max_pitch=pi/8, retarget=20) → Iterable[Vector]
```

Returns a random 3D path as iterable of `Vector` objects.

**Parameters**

- **steps** – count of vertices to generate
- **max_step_size** – max step size
- **max_heading** – limit heading angle change per step to ± max_heading/2, rotation about the z-axis in radians
- **max_pitch** – limit pitch angle change per step to ± max_pitch/2, rotation about the x-axis in radians
- **retarget** – specifies steps before changing global walking target

### 5.7.6 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 Vector.

ezdxf.render.forms.circle(count: int, radius: float = 1, elevation: float = 0, close: bool = False) → Iterable[Vector]

Create polygon vertices for a circle with radius and count corners, 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 Vector objects

ezdxf.render.forms.square(size: float = 1.) → Tuple[Vector, Vector, Vector, Vector]

Returns 4 vertices for a square with a side length of size, lower left corner is (0, 0), upper right corner is (size, size).

ezdxf.render.forms.box(sx: float = 1., sy: float = 1.) → Tuple[Vector, Vector, Vector, Vector]

Returns 4 vertices for a box sx by sy, lower left corner is (0, 0), upper right corner is (sx, sy).

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[Vector]

Create polygon vertices for an ellipse with rx as x-axis radius and ry for y-axis radius with 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 .. 2*pi
- **end_param** – end of ellipse in range 0 .. 2*pi
- **elevation** – z-axis for all vertices

**Returns** vertices in counter clockwise orientation as Vector objects

```python
ezdxf.render.forms.euler_spiral(count: int, length: float = 1, curvature: float = 1, elevation: float = 0) → Iterable[Vector]
```

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 Vector objects

```python
ezdxf.render.forms.ngon(count: int, length: float = None, radius: float = None, rotation: float = 0., elevation: float = 0., close: bool = False) → Iterable[Vector]
```

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 higher priority.

**Parameters**

- **count** – spike count >= 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 Vector objects

```python
ezdxf.render.forms.star(count: int, r1: float, r2: float, rotation: float = 0., elevation: float = 0., close: bool = False) → Iterable[Vector]
```

Returns corner vertices for star shapes.

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.

**Parameters**

- **count** – spike count >= 3
- **r1** – radius 1

Chapter 5. Contents
ezdxf Documentation, Release 0.13

• \textit{r2} – radius 2
• \textit{rotation} – rotation angle in radians
• \textit{elevation} – z-axis for all vertices
• \textit{close} – yields first vertex also as last vertex if \textit{True}.

\textbf{Returns} vertices as \textit{Vector} objects

\texttt{ezdxf.render.forms.gear(count: int, top_width: float, bottom_width: float, height: float, outside_radius: float, elevation: float = 0, close: bool = False) \rightarrow Iterable[Vector]}

\textbf{Returns} \textit{gear (cogwheel)} corner vertices.

\textbf{Warning:} This function does not create correct gears for mechanical engineering!

\textbf{Parameters}

- \textit{count} – teeth count \( \geq 3 \)
- \textit{top_width} – teeth width at outside radius
- \textit{bottom_width} – teeth width at base radius
- \textit{height} – teeth height; base radius = outside radius - height
- \textit{outside_radius} – outside radius
- \textit{elevation} – z-axis for all vertices
- \textit{close} – yields first vertex also as last vertex if \textit{True}.

\textbf{Returns} vertices in counter clockwise orientation as \textit{Vector} objects

\textbf{3D Forms}

Create 3D forms as \textit{MeshTransformer} objects.

\texttt{ezdxf.render.forms.cube(center: bool = True) \rightarrow MeshTransformer}

Create a cube as \textit{MeshTransformer} object.

\textbf{Parameters} \textit{center} – ‘mass’ center of cube, \((0, 0, 0)\) if \textit{True}, else first corner at \((0, 0, 0)\)

\textbf{Returns:} \textit{MeshTransformer}

\texttt{ezdxf.render.forms.cylinder(count: int, radius: float = 1., top_radius: float = None, top_center: Vertex = (0, 0, 1), caps=True, ngons=True) \rightarrow MeshTransformer}

Create a cylinder as \textit{MeshTransformer} object, the base center is fixed in the origin \((0, 0, 0)\).

\textbf{Parameters}

- \textit{count} – profiles edge count
- \textit{radius} – radius for bottom profile
- \textit{top_radius} – radius for top profile, if \textit{None} \textit{top_radius} == radius
- \textit{top_center} – location vector for the center of the top profile
- \textit{caps} – close hull with bottom cap and top cap (as N-gons)
- \textit{ngons} – use ngons for caps if \textit{True} else subdivide caps into triangles

5.7. Rendering
Returns: `MeshTransformer`

ezdf.x.render.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`

New in version 0.11.

nezdf.x.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`

New in version 0.11.

nezdf.x.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`

New in version 0.11.

nezdf.x.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

New in version 0.11.

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 clock wise 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*

### 5.7.7 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.

```python
class ezdxf.render.MeshBuilder
```

**vertices**
List of vertices as *Vector* 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.

```python
copy()
   Returns a copy of mesh.
```

```python
faces_as_vertices() \rightarrow\text{Iterable}[\text{List}[\text{Vector}]]
   Iterate over all mesh faces as list of vertices.
```

```python
edges_as_vertices() \rightarrow\text{Iterable}[\text{Tuple}[\text{Vector}, \text{Vector}]]
   Iterate over all mesh edges as tuple of two vertices.
```

```python
add_vertices(vertices: \text{Iterable}[\text{Vertex}]) \rightarrow\text{Sequence}[\text{int}]
   Add new vertices to the mesh, each vertex is a (*x*, *y*, *z*) tuple or a *Vector* 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 *Vector* objects

   **Returns**
   - indices of the *vertices* added to the *vertices* list

   **Return type**
   - tuple
```

```python
add_edge(vertices: \text{Iterable}[\text{Vertex}]) \rightarrow\text{None}
   An edge consist of two vertices [*v1*, *v2*], each vertex is a (*x*, *y*, *z*) tuple or a *Vector* 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)}]
```

```python
add_face(vertices: \text{Iterable}[\text{Vertex}]) \rightarrow\text{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 *Vector* 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 Vector 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 (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.

New in version 0.11.1.

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.

New in version 0.12.

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*.

**classmethod from_polyface**(other: Union[Polymesh, Polyface]) → MeshBuilder

Create new mesh from a *Polyface* or *Polymesh* object.

New in version 0.11.1.

**classmethod from_builder**(other: MeshBuilder)

Create new mesh from other mesh builder, faster than *from_mesh()* but supports only *MeshBuilder* and inherited classes.

### 5.7.8 MeshTransformer

Same functionality as *MeshBuilder* but supports inplace transformation.

**class 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 Vector
- **angle** – rotation angle in radians

5.7.9 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** ezdxfr.render.MeshVertexMerger \((precision: int = 6)\)

**Subclass of** MeshBuilder

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

### 5.7.10 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 comparison to MeshVertexMerger and this class also does not support transformations.

```python
class ezdxf.render.MeshAverageVertexMerger(precision: int = 6)
```

Subclass of MeshBuilder

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 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

### 5.8 Add-ons

#### 5.8.1 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
If using FIXED-TABLES, following predefined text styles are available:

- OpenSans
- OpenSansCondensed-Light

New in version 0.12: Write Binary DXF files.

**Tutorial**

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="MIDDLE_CENTER")
```

A simple example of writing really many entities in a short time:
from random import random
from ezdxf.addons import r12writer

MAX_XCOORD = 1000.0
MAX_YCOORD = 1000.0
CIRCLE_COUNT = 1000000

with r12writer("many_circles.dxf") as dxf:
    for i in range(CIRCLE_COUNT):
        dxf.add_circle((MAX_XCOORD*random(), MAX_YCOORD*random()), radius=2)

Show all available line types:

import ezdxf

LINETYPES = [
    'CONTINUOUS', 'CENTER', 'CENTERX2', 'CENTER2',
    'DASHED', 'DASHEDX2', 'DASHED2', 'PHANTOM', 'PHANTOMX2',
    'PHANTOM2', 'DASHDOT', 'DASHDOTX2', 'DASHDOT2', 'DOT',
    'DOTT2', 'DOT', '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')

Reference

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.

New in version 0.12: 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 = '0', color: int = None, linetype = None, 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.

```python
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()`

```python
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()`

```python
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()`

---

5.8. Add-ons
**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()**

**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.

**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
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
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:

```python
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](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()`

```python
add_polymesh(vertices: Iterable[Sequence[float]], size: Tuple[int, int], closed=(False, False), 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](https://github.com/mozman/ezdxf/blob/master/examples/r12writer.py)

**Parameters**

- **vertices** – iterable of `(x, y, z)` tuples, in column order

5.8. Add-ons
• **size** – mesh dimension as \((m, n)\)-tuple, requirement: \(\text{len}(\text{vertices}) == m \times n\)

• **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.

### 5.8.2 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.
from ezdxf.addons import iterdxf

doc = iterdxf.opendxf('big.dxf')
line_exporter = doc.export('line.dxf')
text_exporter = doc.export('text.dxf')
polyline_exporter = doc.export('polyline.dxf')
try:
    for entity in doc.modelspace():
        if entity.dxftype() == 'LINE':
            line_exporter.write(entity)
        elif entity.dxftype() == 'TEXT':
            text_exporter.write(entity)
        elif entity.dxftype() == 'POLYLINE':
            polyline_exporter.write(entity)
finally:
    line_exporter.close()
    text_exporter.close()
    polyline_exporter.close()
doc.close()

Supported DXF types:
3DFACE, ARC, ATTDEF, 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:

newdoc = ezdxf.new()
msp = newdoc.modelspace()
# line is an entity from a big source file
msp.add_foreign_entity(line)
# and so on ...
msp.add_foreign_entity(lwpolyline)
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:

from ezdxf.render import MeshTransformer

# mesh is MESH from a big source file
t = MeshTransformer.from_mesh(mesh)
# create a new MESH entity from MeshTransformer
t.render(msp)

# polyface is POLYFACE from a big source file
t = MeshTransformer.from_polyface(polyface)
# create a new POLYMESH entity from MeshTransformer
t.render_polyface(msp)

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.
ezdxf.addons.iterdxf.opendxf (filename: str) → 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.

```python
ezdxf.addons.iterdxf.modelspace(filename: str, types: Iterable[str]=None) → Iterable[DXFGraphic]
```

Iterate over all modelspace entities as `DXFGraphic` objects of a seekable file.

Use this function to ‘quick’ iterate 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.

```python
ezdxf.addons.iterdxf.single_pass_modelsace(stream: BinaryIO, types: Iterable[str]=None) → 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.

### class ezdxf.addons.iterdxf.IterDXF

**export**(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

```python
ezdxf.addons.iterdxf.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.

```python
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.

```python
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.

### 5.8.3 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

doc = ezdxf.readfile('original.dxf')
tdoc = ezdxf.new()

importer = Importer(doc, tdoc)

# import all entities from source modelspace into modelspace of the target drawing
importer.import_modelsace()

# 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
```

(continues on next page)
ents = sdoc.modelspace().query('CIRCLE LINE')
import.import_entities(ents, tblock)

# 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 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

```python
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

```python
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

```python
import_modelsace(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

```python
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

```python
import_paperspace_layouts() → 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.

```python
import_table(name: str, entries: Union[str, Iterable[str]] = '*', replace=False) → None
```

Import specific table entries from source drawing into target drawing.
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

**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

**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

### 5.8.4 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 scripts 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 matplotlib.pyplot as plt
import ezdxf
from ezdxf.addons.drawing import RenderContext, Frontend
from ezdxf.addons.drawing.matplotlib_backend import MatplotlibBackend

doc = ezdxf.readfile('your.dxf')
msp = doc.modelspace()

fig = plt.figure()
ax = fig.add_axes([0, 0, 1, 1])
ctx = RenderContext(doc)
out = MatplotlibBackend(ax)
Frontend(ctx, out).draw_layout(layout, finalize=True)
fig.savefig('your.png', dpi=300)
```

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 in-
terface: ezdxf.addons.drawing.backend_interface.DrawingBackend. Currently a PyQt5 (QGraphicsScene based) and Matplotlib backend 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 ezdxf.addons.drawing.colors.ColorContext class supports passing a 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**

- Patterns, line styles and line widths are ignored
- 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 leaders
- 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)
  - hatches with holes (holes are rendered filled)
  - 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**

- render polylines using draw_line_string if it does not contain any arcs
- pass the font to backend if available
- deal with nested polygons/hatches by triangulating them: Triangulation
- both the matplotlib and pyqt backends have built-in support for rendering hatched patterns (see MatplotlibHatch and QtBrushHatch) so the interface could pass that information through or query the backend to determine whether it automatically supports complex drawing commands such as hatching, or whether the frontend should break the shape into simpler commands (i.e. calculate and draw each line of a hatch)
• 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

5.8.5 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('

    ')
```


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.

5.8.6 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
ezdxf.addons.acadctb.load(filename: str) → Union[ColorDependentPlotStyles, NamedPlotStyles]
Load the CTB or STB file `filename` from file system.

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.

New in version 0.10.

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, linetline_size, linetype, adaptive_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

```python
def 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

```python
save(filename: str) -> None
```

Save CTB file as `filename` to the file system.

```python
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 ezdxfs.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`

- **__getitem__**(name: str) -> PlotStyle
  Returns `PlotStyle` by `name`.

- **__delitem__**(name: str)
  Delete plot style `name`. Plot style 'Normal' is not deletable.

- **__iter__**()
  Iterable of all plot style names.

- **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

- **get_lineweight**(name: str)
  Returns the assigned lineweight for `PlotStyle name` in millimeter.
get_lineweight_index \((\text{lineweight: float}) \rightarrow \text{int}\)
Get index of \text{lineweight} in the lineweight table or append \text{lineweight} to lineweight table.

get_table_lineweight \((\text{index: int}) \rightarrow \text{float}\)
Returns lineweight in millimeters of lineweight table entry \text{index}.

- **Parameters**
  - \text{index} – lineweight table index = \text{PlotStyle.linewidth}
  - **Returns**
    - \text{lineweight} in mm or 0.0 for use entity lineweight

set_table_lineweight \((\text{index: int, lineweight: float}) \rightarrow \text{int}\)
Argument \text{index} is the lineweight table index, not the AutoCAD Color Index (ACI).

- **Parameters**
  - \text{index} – lineweight table index = \text{PlotStyle.linewidth}
  - \text{lineweight} – in millimeters

save \((\text{filename: str}) \rightarrow \text{None}\)
Save STB file as \text{filename} to the file system.

write \((\text{stream: BinaryIO}) \rightarrow \text{None}\)
Compress and write STB file to binary \text{stream}.

**PlotStyle**

```python
class ezdxf.addons.acadctb.PlotStyle
```

- **index**
  - Table index (0-based). (int)
- **aci**
  - \text{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 \text{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 \text{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 \text{dithering} option must be active.
- **linetype**
  - Overrides the entity linetype, default value is \text{OBJECT_LINETYPE}. (bool)
- **adaptive_linetype**
  - True if a complete linetype pattern is more important than a correct linetype scaling, default is \text{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 FILL_STYLE_OBJECT (int)

dithering
Depending on the capabilities of your plotter, dithering approximates the colors with dot patterns. When this option is 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

ezdxf.addons.acadctb.AUTOMATIC
ezdxf.addons.acadctb.OBJECT_LINEWEIGHT
ezdxf.addons.acadctb.OBJECT_LINETYPE
ezdxf.addons.acadctb.OBJECT_COLOR
ezdxf.addons.acadctb.OBJECT_COLOR2

Line End Style

<table>
<thead>
<tr>
<th>End Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>END_STYLE_BUTT</td>
<td>0</td>
</tr>
<tr>
<td>END_STYLE_SQUARE</td>
<td>1</td>
</tr>
<tr>
<td>END_STYLE_ROUND</td>
<td>2</td>
</tr>
<tr>
<td>END_STYLE_DIAMOND</td>
<td>3</td>
</tr>
<tr>
<td>ENDSTYLEOBJECT</td>
<td>4</td>
</tr>
</tbody>
</table>

Line Join Style

<table>
<thead>
<tr>
<th>Join Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOINSTYLEMITER</td>
<td>0</td>
</tr>
<tr>
<td>JOINSTYLEBEVEL</td>
<td>1</td>
</tr>
<tr>
<td>JOINSTYLEROUND</td>
<td>2</td>
</tr>
<tr>
<td>JOINSTYLEDIAMOND</td>
<td>3</td>
</tr>
<tr>
<td>JOINSTYLEOBJECT</td>
<td>5</td>
</tr>
</tbody>
</table>
Fill Style

- Fill style: Use entity fill style
  - Solid
  - Checkerboard
  - Crosshatch
  - Diamonds
  - Horizontal Bars
  - Slant Left
  - Slant Right
  - Square Dots
  - Vertical Bars

- 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

- Solid
- Dashed
- Dotted
- Dash Dot
- Short Dash
- Medium Dash
- Long Dash
- Short Dash x2
- Medium Dash x2
- Long Dash x2
- Medium Long Dash
- Medium Dash Short Dash Short Dash
- Long Dash Short Dash
- Long Dash Dot Dot
- Long Dash Dot
- Long Dash Dot
- Medium Dash Dot Short Dash Dot
- Sparse Dot
- ISO Dash
### Linetype

<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>

#### 5.8.7 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.

New in version 0.11.

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
cube1 = 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:

```python
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.

```python
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()
```

(continues on next page)
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()

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:

```
A.union(B)
```

___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:

```
A.subtract(B)
```

___sub__ (other: CSG) → CSG

```
difference = A - B
```

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)
```
ezdxf Documentation, Release 0.13

(continued from previous page)

| __mul__ (other: CSG) → CSG


text = 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.

5.8.8 Showcase Forms

MengerSponge

Build a 3D Menger sponge.

class ezdxf.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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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 OXO</td>
</tr>
<tr>
<td>3</td>
<td>Jerusalem Cube</td>
</tr>
</tbody>
</table>

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

```python
cubes() → Iterable[ezdxf.render.mesh.MeshTransformer]
Yields all cubes of the menger sponge as individual MeshTransformer objects.

mesh() → ezdxf.render.mesh.MeshTransformer
Returns geometry as one MeshTransformer object.
```

Menger Sponge `kind=0`:

---

Menger Sponge `kind=1`:
Menger Sponge $\text{kind}=2$: 
Jerusalem Cube \texttt{kind=3}:
SierpinskyPyramid

Build a 3D Sierpinsky Pyramid.

class ezdxf.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:
5.8.9 ODA File Converter Support

Use an installed ODA File Converter for converting between different versions of .dwg, .dxb and .dxf.

**Warning:** Execution of an external application is a big security issue! Especially when the path to the executable can be altered.

To avoid this problem delete the `ezdxfall.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 later, Mac OS X, and Linux in 32/64-bit RPM and DEB format.

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:
Usage:

```python
from ezdxf.addons import odafc

# Load a DWG file
doc = odafc.readfile('my.dwg')

# Use loaded document like any other ezdxf document
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.

## 5.9 DXF Internals

- **DXF Reference** provided by Autodesk.
- **DXF Developer Documentation** provided by Autodesk.

### 5.9.1 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)</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>
<tr>
<td>Group Code</td>
<td>Value Type</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------------------</td>
</tr>
<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>
<tr>
<td>Group Code</td>
<td>Meaning</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
</tr>
<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 ezdx 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. `\U'\TEST\U+7F3A\U+4E4F\U+89E3\U+91CA\U+6B63THIS\U+56FE\U+')

To support the DXF unicode encoding ezdxf registers an encoding codec dxf_backslash_replace, defined in ezdxf.l1ldxf.encoding().
String values can be stored with these dxf group codes:

- 0 - 9
- 100 - 101
- 300 - 309
- 410 - 419
- 430 - 439
- 470 - 479
- 499 - 1003

Multi tag text (MTEXT)

If the text string is less than 250 characters, all characters appear in tag (1, ...). If the text string is longer than 250 characters, the string is divided into 250-character chunks, which appear in one or more (3, ...) tags. If (3, ...) tags are used, the last group is a (1, ...) 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, 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:

```
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:

```
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:

```plaintext
102
{YOURAPPID
...
102
YOURAPPID}
```

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](#).

**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 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. Polylines 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. **ACDSDATA**: (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
SECTION 2 BLOCKS
(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://bitbucket.org/mozman/ezdxf/downloads/Minimal_DXF_AC1021.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 (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 its getting complicated!

Most information about the OBJECTS section is just guessed or gathered by trial 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

(continues on next page)
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
5.9.2 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 `ezdxfl 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).
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!

```
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  <<- 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 ENDBKL 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
  ...  <<< end of 1. Block definition
ENDBLK
 0   <<< start of 2. BLOCK definition
BLOCK
  ...  <<< Block content
  ...  <<< 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**

```
0
VIEW
2   <<< name of view
VIEWNAME
70  <<< flags bit-coded: 1st bit -> (0/1 = modelspace/paperspace)
 0   <<< modelspace
40  <<< view width in Display Coordinate System (DCS)
20.01
10  <<< view center point in DCS
40.36 <<<   x value
20  <<< group code for y value
15.86 <<<   y value
41  <<< view height in DCS
17.91
11  <<< view direction from target point, 3D vector
 0.0 <<<   x value
21  <<< group code for y value
 0.0 <<<   y value
31  <<< group code for z value
 1.0 <<<   z value
12  <<< target point in WCS
 0.0 <<<   x value
22  <<< group code for y value
 0.0 <<<   y value
32  <<< group code for z value
 0.0 <<<   z value
42  <<< lens (focal) length
 50.0 <<< 50mm
43  <<< front clipping plane, offset from target
 0.0
44  <<< back clipping plane, offset from target
 0.0
50  <<< twist angle
 0.0
71  <<< view mode
0
```

See also:

*Coordinate Systems*
DXF R2000+

Mostly the same structure as DXF R12, but with handle, owner tag and subclass markers.

| 0 | <<< adding the VIEW table head, just for information |
| 2 | <<< table name |
| VIEW | <<< handle of table, see owner tag of VIEW table entry |
| 37C | <<< owner tag of table, always #0 |
| 330 | <<< VIEW table (max.) count, not reliable (ignore) |
| 100 | <<< subclass marker |
| AcDbSymbolTable | |
| 70 | <<< VIEW table (max.) count, not reliable (ignore) |
| 9 | |
| VIEW | <<< first VIEW table entry |
| 5 | <<< handle |
| 3EA | <<< 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 |

(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.

```plaintext
0    <<< table start
TABLE
2    <<< table type
VPORT
70   <<< VPORT table (max.) count, not reliable (ignore)
3    <<< 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
```

(continues on next page)
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
0
10
0.0
20
0.5
11
0.45
21
1.0
12
8.21
22
9.41

...  <<< next VPORT entry
...
DXF R2000+

Mostly the same structure as DXF R12, but with handle, owner tag and subclass markers.

```
0  <<< table start
TABLE
2  <<< table type
VPORT
5  <<< table handle
151F
330  <<< owner, table has no owner - always #0
0
100  <<< subclass marker
AcDbSymbolTable
70  <<< VPOR T table (max.) count, not reliable (ignore)
3
0  <<< first VPOR T entry
VPOR T
5  <<< entry handle
158B
330  <<< owner, VPOR T table is owner of VPOR T entry
151F
100  <<< subclass marker
AcDbSymbolTableRecord
100  <<< subclass marker
AcDbViewportTableRecord
2  <<< VPOR T (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
```
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>y value, virtual coordinates in range [0 - 1]</td>
</tr>
<tr>
<td>11</td>
<td>upper-right corner of viewport</td>
</tr>
<tr>
<td>1.0</td>
<td>x value, virtual coordinates in range [0 - 1]</td>
</tr>
<tr>
<td>21</td>
<td>group code for y value</td>
</tr>
<tr>
<td>1.0</td>
<td>y value, virtual coordinates in range [0 - 1]</td>
</tr>
<tr>
<td>12</td>
<td>view center point (in DCS)</td>
</tr>
<tr>
<td>13.71</td>
<td>x value</td>
</tr>
<tr>
<td>22</td>
<td>group code for y value</td>
</tr>
<tr>
<td>0.38</td>
<td>y value</td>
</tr>
<tr>
<td>13</td>
<td>snap base point (in DCS)</td>
</tr>
<tr>
<td>0.0</td>
<td>x value</td>
</tr>
<tr>
<td>23</td>
<td>group code for y value</td>
</tr>
<tr>
<td>0.0</td>
<td>y value</td>
</tr>
<tr>
<td>14</td>
<td>snap spacing X and Y</td>
</tr>
<tr>
<td>1.0</td>
<td>x value</td>
</tr>
<tr>
<td>24</td>
<td>group code for y value</td>
</tr>
<tr>
<td>1.0</td>
<td>y value</td>
</tr>
<tr>
<td>15</td>
<td>grid spacing X and Y</td>
</tr>
<tr>
<td>0.0</td>
<td>x value</td>
</tr>
<tr>
<td>25</td>
<td>group code for y value</td>
</tr>
<tr>
<td>0.0</td>
<td>y value</td>
</tr>
<tr>
<td>16</td>
<td>view direction from target point (in WCS)</td>
</tr>
<tr>
<td>1.0</td>
<td>x value</td>
</tr>
<tr>
<td>26</td>
<td>group code for y value</td>
</tr>
<tr>
<td>-1.0</td>
<td>y value</td>
</tr>
<tr>
<td>36</td>
<td>group code for z value</td>
</tr>
<tr>
<td>1.0</td>
<td>z value</td>
</tr>
<tr>
<td>17</td>
<td>view target point (in WCS)</td>
</tr>
<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>40</td>
<td>view height</td>
</tr>
<tr>
<td>35.22</td>
<td>viewport aspect ratio</td>
</tr>
<tr>
<td>0.99</td>
<td>lens (focal) length</td>
</tr>
<tr>
<td>50.0</td>
<td>50mm</td>
</tr>
<tr>
<td>43</td>
<td>front clipping planes, offsets from target point</td>
</tr>
<tr>
<td>0.0</td>
<td>back clipping planes, offsets from target point</td>
</tr>
<tr>
<td>50</td>
<td>snap rotation angle</td>
</tr>
<tr>
<td>0.0</td>
<td>view twist angle</td>
</tr>
<tr>
<td>0.0</td>
<td>view mode</td>
</tr>
<tr>
<td>72</td>
<td>circle zoom percent</td>
</tr>
<tr>
<td>1000</td>
<td>fast zoom setting</td>
</tr>
<tr>
<td>1</td>
<td>UCSICON setting</td>
</tr>
<tr>
<td>3</td>
<td>snap on/off</td>
</tr>
</tbody>
</table>

(continues on next page)
76  <<< grid on/off
0
77  <<< snap style
0
78  <<< snap isopair
0
281  <<< render mode 1-6 (... too many options)
0  <<< 0 = 2D optimized (classic 2D)
65  <<< Value of UCSVP for this viewport. (0 = UCS will not change when this
--viewport 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)
LTYPE Table

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

5.9. DXF Internals
### 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
<<< 2. LTYPE table entry
LTYPE
<<< LTYPE data tags and so on
0 <<< end of LTYPE table
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
<<< 2. LTYPE table entry
LTYPE
<<< LTYPE data tags and so on
0 <<< end of LTYPE table
ENDTAB
```

### Simple Line Type

`ezdxf` setup for line type ‘CENTER’:

```
dwg.linetypes.new("CENTER", dxfattribs={
    description = "Center ____ _ ____ _ ____ _ ____ _ ____",
    pattern=[2.0, 1.25, -0.25, 0.25, -0.25],
})
```
Simple Line Type Tag Structure DXF R2000+

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>line type table entry</td>
</tr>
<tr>
<td>LTYPE</td>
<td>handle of line type</td>
</tr>
<tr>
<td>5</td>
<td>owner handle, handle of LTYPE table</td>
</tr>
<tr>
<td>1B1</td>
<td></td>
</tr>
<tr>
<td>330</td>
<td></td>
</tr>
<tr>
<td>5F</td>
<td>subclass marker</td>
</tr>
<tr>
<td>100</td>
<td>subclass marker</td>
</tr>
<tr>
<td>100</td>
<td>subclass marker</td>
</tr>
<tr>
<td>2</td>
<td>line type name</td>
</tr>
<tr>
<td>70</td>
<td>flags</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

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=-.1,Y=-.05],-.25',
})
```
TEXT Tag Structure

0
LTYPE
5
614
330
5F
100 <<< subclass marker
AcDbSymbolTableRecord
100 <<< subclass marker
AcDbLinetypeTableRecord
2
GASLEITUNG
70
0
3
Gasleitung2 ----GAS----GAS----GAS----GAS----GAS----GAS--
72
65
73
3
40
1
49
0.5
74
0
49
-0.2
74
2
75
0
340
11
46
0.1
50
0.0
44
-0.1
45
-0.05
9
GAS
49
-0.25
74
0

Complex Line Type SHAPE

ezdx

ezdx setup for line type ‘GRENZE2’:
```python
dwg.linetypes.new('GRENZE2', dxattribs=
    {'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)
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
<<< DIMSTYLE data tags
0 <<< 2. DIMSTYLE table entry
DIMSTYLE
<<< DIMSTYLE data tags and so on
0 <<< end of DIMSTYLE table
ENDTAB
```

DIMSTYLE Entry DXF R12

DIMSTYLE Variables DXF R12

Source: CADDManager 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’[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>

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>DIMCEN</td>
<td>141</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 = No center marks or lines are drawn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• &lt;0 = Centerlines are drawn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• &gt;0 = Center marks are drawn</td>
</tr>
<tr>
<td>DIMCLRD</td>
<td>176</td>
<td>Assigns colors to dimension lines, arrowheads, and dimension leader lines.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 = BYBLOCK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1-255 = ACI AutoCAD Color Index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 256 = BYLAYER</td>
</tr>
<tr>
<td>DIMCLRE</td>
<td>177</td>
<td>Assigns colors to dimension extension lines, values like DIMCLRD</td>
</tr>
<tr>
<td>DIMCLRT</td>
<td>178</td>
<td>Assigns colors to dimension text, values like DIMCLRD</td>
</tr>
<tr>
<td>DIMDLE</td>
<td>46</td>
<td>Sets the distance the dimension line extends beyond the extension line when oblique strokes are drawn instead of arrowheads.</td>
</tr>
<tr>
<td>DIMDLI</td>
<td>43</td>
<td>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.</td>
</tr>
<tr>
<td>DIMEXE</td>
<td>44</td>
<td>Specifies how far to extend the extension line beyond the dimension line.</td>
</tr>
<tr>
<td>DIMEXO</td>
<td>42</td>
<td>Specifies how far extension lines are offset from origin points. With fixed-length extension lines, this value determines the minimum offset.</td>
</tr>
</tbody>
</table>
### 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 \times (\text{DIMASZ} + \text{DIMGAP})).</td>
</tr>
<tr>
<td>DIMLFAC</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 DIMLFAC before being converted to dimension text. Positive values of DIMLFAC are applied to dimensions in both modelspace and paperspace; negative values are applied to paperspace only. DIMLFAC applies primarily to nonassociative dimensions (DIMASSOC set 0 or 1). For nonassociative dimensions in paperspace, DIMLFAC must be set individually for each layout viewport to accommodate viewport scaling. DIMLFAC has no effect on angular dimensions, and is not applied to the values held in DMRND, DIMTM, or DIMTP.</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>
</table>
| DIMSCALE | 40   | 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.  
• 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.  
• >0 = A scale factor is computed that leads text sizes, arrowhead sizes, and other scaled distances to plot at their face values.  
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. |
| DIMSE1  | 75   | Suppresses display of the first extension line.  
• 0 = Extension line is not suppressed  
• 1 = Extension line is suppressed |
| DIMSE2  | 76   | Suppresses display of the second extension line.  
• 0 = Extension line is not suppressed  
• 1 = Extension line is suppressed |

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>
| DIMSOXD  | 175  | Suppresses arrowheads if not enough space is available inside the extension lines.  
|          |      | • 0 = Arrowheads are not suppressed  
|          |      | • 1 = Arrowheads are suppressed  
|          |      | 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. |
| DIMITAD   | 77   | Controls the vertical position of text in relation to the dimension line.  
|          |      | • 0 = Centers the dimension text between the extension lines.  
|          |      | • 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 (DIMITIH = 1). The distance from the dimension line to the baseline of the lowest line of text is the current DIMGAP value.  
|          |      | • 2 = Places the dimension text on the side of the dimension line farthest away from the defining points.  
|          |      | • 3 = Places the dimension text to conform to Japanese Industrial Standards (JIS).  
|          |      | • 4 = Places the dimension text below the dimension line. |
| DIMITFAC  | 146  | Specifies a scale factor for the text height of fractions and tolerance values relative to the dimension text height, as set by DIMITXT.  
|          |      | 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. |

Continued on next page
<table>
<thead>
<tr>
<th>DIMVAR</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
</table>
| DIMTH | 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 |
| DIMTX | 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, DIMTX 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 4 – continued from previous 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  
5F  
330 <<< owner tag, tables has no owner  
0  

(continues on next page)
100 <<< subclass marker
AcDbSymbolTable
70 <<< count of dimension styles defined in this table, AutoCAD ignores this value
9
0 <<< 1. DIMSTYLE table entry
DIMSTYLE
0 <<< DIMSTYLE data tags
0 <<< 2. DIMSTYLE table entry
DIMSTYLE
0 <<< DIMSTYLE data tags and so on
0 <<< end of DIMSTYLE table
ENDTAB

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>

5.9. DXF Internals
Additional DIMSTYLE 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>

### 5.9. DXF Internals

<table>
<thead>
<tr>
<th>DIMUNIT</th>
<th>278</th>
<th>Sets units for all dimension types except Angular.</th>
</tr>
</thead>
<tbody>
<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>
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 \texttt{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</td>
</tr>
<tr>
<td>DIMJOGANG</td>
<td>50</td>
<td>Angle of oblique dimension line segment in jogged radius dimension</td>
</tr>
<tr>
<td>DMLTYPE_HANDLE</td>
<td>345</td>
<td>Specifies the LINETYPE of the dimension line. Handle to LTYPE table entry</td>
</tr>
<tr>
<td>DMLTEX1_HANDLE</td>
<td>346</td>
<td>Specifies the LINETYPE of the extension line 1. Handle to LTYPE table entry</td>
</tr>
<tr>
<td>DMLTEX2_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 <<< group code, which differs from R2007 DMLTYPE
1005 FFFF <<< handle to LTYPE entry
1001 ACAD_DSTYLE_DIM_EXT1_LINETYPE <<< linetype for extension line 1
1070 381 <<< group code, which differs from R2007 DMLTEX1
1005 FFFF <<< handle to LTYPE entry
1001 ACAD_DSTYLE_DIM_EXT2_LINETYPE <<< linetype for extension line 1
1070 382 <<< group code, which differs from R2007 DMLTEX2
1005 FFFF <<< handle to LTYPE entry
1001 ACAD_DSTYLE_DIMEXT_ENABLED <<< extension line fixed
1070 383 <<< group code, which differs from R2007 DIMEXFIX
1070
```
(continues on next page)
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 Entity**

See also:
- DXF Reference: DIMENSION
- DXFInternals: DIMSTYLE Table

**DXF Objects**

TODO

**5.9.3 Management Structures**

**Block Management Structures**

A BLOCK is a kind of layout like the modelspace or a paperspace layout, with the similarity that all these layouts are containers for other 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, and I haven’t tested what happens I the second name is different to the first block name.

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 (LT 2018) 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:

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;&lt;&lt; start of a SECTION</td>
</tr>
<tr>
<td>2</td>
<td>&lt;&lt;&lt; start of BLOCKS section</td>
</tr>
<tr>
<td>8</td>
<td>&lt;&lt;&lt; layer, what this layer definition does is another fact, I don’t know</td>
</tr>
<tr>
<td>0</td>
<td>(now)</td>
</tr>
<tr>
<td>2</td>
<td>&lt;&lt;&lt; block name ArchTick</td>
</tr>
<tr>
<td>70</td>
<td>&lt;&lt;&lt; flags</td>
</tr>
<tr>
<td>10</td>
<td>&lt;&lt;&lt; base point, x 0.0</td>
</tr>
<tr>
<td>20</td>
<td>&lt;&lt;&lt; base point, y 0.0</td>
</tr>
<tr>
<td>30</td>
<td>&lt;&lt;&lt; base point, z 0.0</td>
</tr>
<tr>
<td>3</td>
<td>&lt;&lt;&lt; second BLOCK name, same as (2, name) ArchTick</td>
</tr>
</tbody>
</table>

(continues on next page)
Block Definitions in DXF R2000 and later

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 a kind of 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. 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 entities of the model space and the active paperspace 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!
```
DXF R13 BLOCK_RECORD structure:

0                  <<< start of a SECTION
SECTION            2                  <<< start of TABLES section
TABLES

5.9. DXF Internals
Table of Contents

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>416</td>
</tr>
</tbody>
</table>
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 by the INSERT entity.

All layouts have at least a BLOCK definition in the BLOCKS section and since DXF R13 exists 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 paper space BLOCK is *Paper_Space (DXF R12: $PAPER_SPACE), the entities of these two layouts are stored in the ENTITIES section, DXF R12 supports just one paperspace layout.

DXF R13 and later 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_Space1, the second *Paper_Space2 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.

5.9. DXF Internals
360
1C3
102
102 <<< reactor (required?)

ACAD_REACTORS
330
1A <<< pointer to "ACAD_LAYOUT" DICTIONARY (layout management table)
102
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
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
1.0
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)

(continues on next page)
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 PSLTSCALE 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
-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)
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. And the most values
0
VIEWPORT
5     <<< handle
1B4
102    <<< extension dictionary (ignore)
(ACAD_XDICTIONARY
360
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

(continues on next page)
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>z value</td>
<td>0.0</td>
</tr>
<tr>
<td>width in paperspace units</td>
<td>40</td>
</tr>
<tr>
<td>VIEW size in AutoCAD, depends on the workstation configuration</td>
<td>23.55</td>
</tr>
<tr>
<td>height in paperspace units</td>
<td>9.00</td>
</tr>
<tr>
<td>VIEW size in AutoCAD, depends on the workstation configuration</td>
<td>68</td>
</tr>
<tr>
<td>viewport status field -1/0/n</td>
<td>2</td>
</tr>
<tr>
<td>&gt;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</td>
<td>69</td>
</tr>
<tr>
<td>viewport ID</td>
<td>1</td>
</tr>
<tr>
<td>&quot;main&quot; viewport has always ID=1</td>
<td>12</td>
</tr>
<tr>
<td>view center point in Drawing Coordinate System (DCS), defines the center point of the VIEW in relation to the LAYOUT origin</td>
<td>5.25</td>
</tr>
<tr>
<td>group code for x value</td>
<td>22</td>
</tr>
<tr>
<td>y value</td>
<td>4.00</td>
</tr>
<tr>
<td>snap base point in modelspace</td>
<td>0.0</td>
</tr>
<tr>
<td>group code for x value</td>
<td>23</td>
</tr>
<tr>
<td>y value</td>
<td>0.0</td>
</tr>
<tr>
<td>snap spacing in modelspace units</td>
<td>0.5</td>
</tr>
<tr>
<td>group code for x value</td>
<td>24</td>
</tr>
<tr>
<td>y value</td>
<td>0.5</td>
</tr>
<tr>
<td>grid spacing in modelspace units</td>
<td>0.5</td>
</tr>
<tr>
<td>group code for x value</td>
<td>25</td>
</tr>
<tr>
<td>y value</td>
<td>0.5</td>
</tr>
<tr>
<td>view direction vector from target (in WCS)</td>
<td>0.0</td>
</tr>
<tr>
<td>group code for x value</td>
<td>26</td>
</tr>
<tr>
<td>y value</td>
<td>0.0</td>
</tr>
<tr>
<td>group code for z value</td>
<td>36</td>
</tr>
<tr>
<td>z value</td>
<td>1.0</td>
</tr>
<tr>
<td>view target point</td>
<td>17</td>
</tr>
<tr>
<td>group code for x value</td>
<td>0.0</td>
</tr>
<tr>
<td>y value</td>
<td>27</td>
</tr>
<tr>
<td>group code for y value</td>
<td>0.0</td>
</tr>
<tr>
<td>z value</td>
<td>37</td>
</tr>
<tr>
<td>group code for z value</td>
<td>36</td>
</tr>
<tr>
<td>perspective lens length, focal length?</td>
<td>42</td>
</tr>
<tr>
<td>50mm</td>
<td>50.0</td>
</tr>
<tr>
<td>front clip plane z value</td>
<td>43</td>
</tr>
<tr>
<td>z value</td>
<td>0.0</td>
</tr>
<tr>
<td>back clip plane z value</td>
<td>44</td>
</tr>
<tr>
<td>z value</td>
<td>0.0</td>
</tr>
<tr>
<td>view height (in modelspace units)</td>
<td>45</td>
</tr>
<tr>
<td>9.00</td>
<td>9.00</td>
</tr>
<tr>
<td>snap angle</td>
<td>70</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>view twist angle</td>
<td>71</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>circle zoom percent</td>
<td>1000</td>
</tr>
<tr>
<td>Viewport status bit-coded flags (... too many options)</td>
<td>90</td>
</tr>
<tr>
<td>b11001000000000100000</td>
<td>819232</td>
</tr>
<tr>
<td>plot style sheet name assigned to this viewport</td>
<td>1</td>
</tr>
</tbody>
</table>
281 <<< render mode (... too many options)
0 <<< 0 = 2D optimized (classic 2D)
71 <<< UCS per viewport flag
1 <<< 1 = This viewport stores its own UCS which will become the current UCS whenever the viewport is activated
74 <<< Display UCS icon at UCS origin flag
0 <<< this field is currently being ignored and the icon always represents the viewport UCS
110 <<< UCS origin (3D point)
0.0 <<< x value
0.0 <<< y value
0.0 <<< z value
111 <<< UCS X-axis (3D vector)
1.0 <<< x value
0.0 <<< y value
0.0 <<< z value
112 <<< UCS Y-axis (3D vector)
0.0 <<< x value
1.0 <<< y value
0.0 <<< z value
79 <<< Orthographic type of UCS (... too many options)
0 <<< 0 = UCS is not orthographic
146 <<< elevation
0.0
170 <<< shade plot mode (0/1/2/3 for as displayed/wireframe/hidden/rendered)
0 <<< as displayed
61 <<< frequency of major grid lines compared to minor grid lines
5 <<< major grid subdivided by 5
348 <<< visual style ID/handle (optional)
9F
292 <<< default lighting flag, on when no user lights are specified.
1
282 <<< Default lighting type (0/1 = one distant light/two distant lights)
1 <<< one distant light
141 <<< view brightness
0.0
142 <<< view contrast
0.0
63 <<< ambient light color (ACI), write only if not black color
250
421 <<< ambient light color (RGB), write only if not black color
3355443

5.10 Developer Guides

Information about ezdxf internals.
5.10.1 Internal Data Structures

Entity Database

The `EntityDB` is a simple key/value database to store `DXFEntity` objects by its 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 (group code 5 or 105).

```python
class ezdxf.entitydb.EntityDB

    __getitem__(handle: str) → DXFEntity
    Get entity by handle.

    __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.

    get(handle: str) → Optional[DXFEntity]
    Returns entity for handle or None if no entry for handle exist.

    next_handle() → str
    Returns next unique handle.

    keys() → Iterable[str]
    Iterable of all handles.

    values() → Iterable[DXFEntity]
    Iterable of all entities.

    items() → Iterable[Tuple[str, DXFEntity]]
    Iterable of all (handle, entities) pairs.

    add(entity: DXFEntity) → None
    Add entity to database, assigns a new handle to the entity if `entity.dxf.handle` is None.

    delete_entity(entity: DXFEntity) → None
    Removes entity from database and destroys the entity.

    duplicate_entity(entity: DXFEntity) → DXFEntity
    Duplicates `entity` and its sub entities (VERTEX, ATTRIB, SEQEND) and store them with new handles in the drawing database. This is the recommend method to duplicate DXF entities in a drawing. Graphical entities have to be added to a layout by `add_entity()`, for other DXF entities: DON’T DUPLICATE THEM.

    To import DXF entities into another drawing use the `Importer` add-on.

    An existing owner tag is not changed because this is not the domain of the `EntityDB` class, will be set by adding the duplicated entity to a layout.
```
This is not a deep copy in the meaning of Python, because handles and links are changed.

Entity Space

class ezdxf.entitydb.EntitySpace(entities=None)

An EntitySpace is a collection of 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__() → Iterable[DXFEntity]
Iterate over all entities.

__getitem__(index) → 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].

__len__() → int
Count of entities.

has_handle(handle: str) → bool
True if handle is present.

purge() Remove deleted entities.

reorder(order: int = 1) → None
Reorder entities in place.

Parameters order: 1 = priority order (highest first), 2 = z-order (inverted priority, lowest first)

add(entity: DXFEntity) → None
Add entity.

extend(entities: Iterable[DXFEntity]) → None
Add multiple entities.

remove(entity: DXFEntity) → 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(itearable: 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 implementation - don’t change it.

- **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)

- **Methods**
  - `__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)'.
  - `clone() → ezdxf.lldxf.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\nLINE\n'
**DXFBinaryTag**

```python
class ezdxf.lldxf.types.DXFBinaryTag(DXFTag)
    Immutable BinaryTags class - immutable by design, not by implementation - don’t change it.

    dxfstr() \rightarrow str
    Returns the DXF string for all vertex components.

    tostring() \rightarrow str
    Returns binary value as single hex-string.
```

**DXFVertex**

```python
class ezdxf.lldxf.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() \rightarrow str
    Returns the DXF string for all vertex components.

    dxftags() \rightarrow Iterable[Tuple]
    Returns all vertex components as single DXFTag objects.
```

**NONE_TAG**

```python
ezdxf.lldxf.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 ezdxf.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) \rightarrow Tags
        Constructor from DXF string.

    dxftype() \rightarrow str
        Returns DXF type of entity, e.g. 'LINE'.

    get_handle() \rightarrow str
        Get DXF handle. Raises DXFValueError if handle not exist.

        Returns handle as plain hex string like 'FF00'
        Raises DXFValueError – no handle found
```

5.10. Developer Guides
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.lldxftags.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
• int (splitcode) – group code of split tag

class ezdxf.lldxftags.ExtendedTags (tags: Iterable[DXFTag]=None, legacy=False)
    Represents the extended DXF tag structure introduced with DXF R13.

    Args:
tags: iterable of DXFTag
legacy: flag for DXF R12 tags

appdata
    Application defined data as list of Tags

subclasses
    Subclasses as list of Tags

xdata
    XDATA as list of Tags

embedded_objects
    embedded objects as list of Tags

noclass
    Property to access self.subclasses[0].

get_handle () → str
    Returns handle as hex string.

dxftype () → str
    Returns DXF type as string like 'LINE'.
replace_handle(handle: str) → None
Replace handle.

legacy_repair()
Legacy (DXF R12) tags handling and repair.

clone() → ExtendedTags
Shallow copy - linked entities are not duplicated!

ExtendedTags knows nothing about the entity database, and has no access to, so it is not possible for
ExtendedTags to do a deep copy, by also copying linked entities (VERTEX, ATTRIB, SEQEND). To
do a deep copy, go one layer up and use DXFEntity.copy().

flatten_subclasses()
Flatten subclasses in legacy mode (DXF R12).

There exists DXF R12 with subclass markers, technical incorrect but works if reader ignore subclass
marker tags, unfortunately ezdxf, tries to use this subclass markers and therefore R12 parsing by ezdxf
does not work without removing this subclass markers.

This method removes the 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)
Append a new XDATA block.

Assumes that no XDATA block with the same appid already exists:

```
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.

```python
class ezdxflldxf.lldxf.packedtags.TagList (data: Iterable = None)
    Store data in a standard Python list.
    Args:
        data: iterable of DXF tag values.
    values
        Data storage as list.
    clone () → 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 ezdxflldxf.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 ezdxflldxf.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.
## ezdxf Documentation, Release 0.13

__getitem__ (index: int)
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.

clone () → 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.ledxf.tagwriter.TagWriter, code=10)

### 5.10.2 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:
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 $flag$.
New in version 0.9: New feature flag
Changed in version 0.10: The new meaning of $flag$ is ...
Deprecated since version 0.11: $flag$ is obsolete

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

5.11 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* ®.

STB  Named plot style table (*NamedPlotStyles*)

true color  RGB color representation, a combination red, green and blue values to define a color.

5.12 Indices and tables

- genindex
- search
Python Module Index

e
ezdxf, 108
ezdxf.addons, 324
ezdxf.addons.acadctb, 339
ezdxf.addons.odafc, 361
ezdxf.addons.pycsg, 348
ezdxf.addons.r12writer, 324
ezdxf.comments, 302
ezdxf.drawing, 111
ezdxf.entities, 174
ezdxf.entities.dxffields, 172
ezdxf.entitydb, 424
ezdxf.layouts, 151
ezdxf.lldxf.extendedtags, 429
ezdxf.lldxf.packedtags, 431
ezdxf.lldxf.tags, 427
ezdxf.lldxf.types, 425
ezdxf.math, 256
ezdxf.options, 301
ezdxf.query, 253
ezdxf.render, 309
ezdxf.render.forms, 314
ezdxf.sections.blocks, 120
ezdxf.sections.classes, 118
ezdxf.sections.entities, 121
ezdxf.sections.header, 117
ezdxf.sections.objects, 121
ezdxf.sections.table, 124
ezdxf.sections.tables, 119
ezdxf.tools, 302

g
guide, 433
Symbols

__abs__() (ezdxf.math.Vector method), 275
__add__() (ezdxf.addons.pycsg.CSG method), 353
__add__() (ezdxf.math.Matrix method), 299
__add__() (ezdxf.math.Vector method), 276
__bool__() (ezdxf.math.Vector method), 276
__contains__() (ezdxf.entities.Dictionary method), 246
__contains__() (ezdxf.entities.dxfgroups.DXFGroup method), 173
__contains__() (ezdxf.entities.dxfgroups.GroupCollection method), 174
__contains__() (ezdxf.entitydb.EntityDB method), 424
__contains__() (ezdxf.layouts.BlockLayout method), 172
__contains__() (ezdxf.layouts.Layout method), 169
__contains__() (ezdxf.layouts.Layouts method), 152
__contains__() (ezdxf.sections.blocks.BlocksSection method), 120
__contains__() (ezdxf.sections.header.HeaderSection method), 117
__contains__() (ezdxf.sections.objects.ObjectsSection method), 122
__contains__() (ezdxf.sections.table.Table method), 124
__copy__() (ezdxf.math.Matrix44 method), 271
__copy__() (ezdxf.math.Vector method), 275
__deepcopy__() (ezdxf.math.Vector method), 275
__delitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__delitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__delitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__delitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__delitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__eq__() (ezdxf.math.Matrix method), 299
__eq__() (ezdxf.math.Vector method), 276
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
__getitem__() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
__getitem__() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
add_3dface() (ezdxf.math.Vector method), 277
add_3dface() (ezdxf.addons.r12writer.R12FastStreamWriter method), 327
add_circle() (ezdxf.addons.r12writer.R12FastStreamWriter method), 327
add_circle() (ezdxf.layouts.BaseLayout method), 167
add_circle() (ezdxf.addons.r12writer.R12FastStreamWriter method), 167
add_class() (ezdxf.sections.classes.ClassesSection method), 118
add_closed_rational_spline() (ezdxf.layouts.BaseLayout method), 161
add_closed_spline() (ezdxf.layouts.BaseLayout method), 160
add_diameter_dim() (ezdxf.layouts.BaseLayout method), 166
add_diameter_dim_2p() (ezdxf.layouts.BaseLayout method), 166
add_dict_var() (ezdxf.entities.Dictionary method), 122
add_dictionary() (ezdxf.sections.objects.ObjectsSection method), 122
add_dictionary_var() (ezdxf.sections.objects.ObjectsSection method), 122
add_dictionary_with_default() (ezdxf.sections.objects.ObjectsSection method), 122
add_edge() (ezdxf.entities.MeshData method), 215
add_edge() (ezdxf.render.MeshBuilder method), 320
add_edge_path() (ezdxf.entities.BoundaryPaths method), 200
add_ellipse() (ezdxf.entities.MeshData method), 202
add_ellipse() (ezdxf.layouts.BaseLayout method), 156
add_entity() (ezdxf.layouts.BaseLayout method), 154
add_extruded_surface() (ezdxf.layouts.BaseLayout method), 168
add_face() (ezdxf.entities.MeshData method), 215
add_face() (ezdxf.render.MeshBuilder method), 320
add_foreign_entity() (ezdxf.layouts.BaseLayout method), 154
add_geodata() (ezdxf.sections.objects.ObjectsSection method), 122
add_hatch() (ezdxf.layouts.BaseLayout method), 161
add_image() (ezdxf.layouts.BaseLayout method), 161
add_image_def() (ezdxf.drawing.Drawing method), 115
add_image_def() (ezdxf.sections.objects.ObjectsSection method), 122
add_import() (ezdxf.addons.dxf2code.Code method), 339
add_leader() (ezdxf.layouts.BaseLayout method), 167
add_line() (ezdxf.addons.dxf2code.Code method), 439
add_line() (ezdxf.addons.r12writer.R12FastStreamWriter method), 326
add_line() (ezdxf.entities.EdgPath method), 202
add_line() (ezdxf.entities.Pattern method), 204
add_line() (ezdxf.layouts.BaseLayout method), 155
add_linear_dim() (ezdxf.layouts.BaseLayout method), 162
add_lines() (ezdxf.addons.dxf2code.Code method), 339
add_lofted_surface () (ezdxf.layouts.BaseLayout method), 168
add_lwpolyline() (ezdxf.layouts.BaseLayout method), 158
add_mesh() (ezdxf.layouts.BaseLayout method), 161
add_mtext() (ezdxf.layout.BaseLayout method), 159
add_multi_point_linear_dim() (ezdxf.layouts.BaseLayout method), 163
add_new_dict() (ezdxf.entities.Dictionary method), 246
add_open_spline() (ezdxf.layouts.BaseLayout method), 160
add_placeholder() (ezdxf.sections.objects.ObjectsSection method), 123
add_point() (ezdxf.addons.r12writer.R12FastStreamWriter method), 327
add_point() (ezdxf.layouts.BaseLayout method), 155
add_polyface() (ezdxf.addons.r12writer.R12FastStreamWriter method), 329
add_polyface() (ezdxf.layouts.BaseLayout method), 158
add_polyline() (ezdxf.addons.r12writer.R12FastStreamWriter method), 329
add_polyline2d() (ezdxf.layouts.BaseLayout method), 158
add_polyline3d() (ezdxf.layouts.BaseLayout method), 158
add_polyline_2d() (ezdxf.addons.r12writer.R12FastStreamWriter method), 328
add polyline_path() (ezdxf.entities.BoundaryPaths method), 200
add_polyline() (ezdxf.addons.r12writer.R12FastStreamWriter method), 329
add_polyline() (ezdxf.layouts.BaseLayout method), 158
add_radius_dim() (ezdxf.layouts.BaseLayout method), 164
add_radius_dim_2p() (ezdxf.layouts.BaseLayout method), 165
add_radius_dim_cra() (ezdxf.layouts.BaseLayout method), 165
add_rational_spline() (ezdxf.layouts.BaseLayout method), 161
add_ray() (ezdxf.layouts.BaseLayout method), 159
add_region() (ezdxf.layouts.BaseLayout method), 167
add_required_classes() (ezdxf.sections.classes.ClassesSection method), 118
add_revolved_surface() (ezdxf.layouts.BaseLayout method), 168
add_shape() (ezdxf.layouts.BaseLayout method), 158
add_solid() (ezdxf.addons.r12writer.R12FastStreamWriter method), 328
add_solid() (ezdxf.layouts.BaseLayout method), 156
add_spline() (ezdxf.entities.EdgPath method), 202
add_spline() (ezdxf.layouts.BaseLayout method), 159
add_spline_control_frame() (ezdxf.layouts.BaseLayout method), 160
add_stack_text() (ezdxf.entities.MText method), 217
add_surface() (ezdxf.layouts.BaseLayout method), 168
add_swept_surface() (ezdxf.layouts.BaseLayout method), 168
add_text() (ezdxf.addons.r12writer.R12FastStreamWriter method), 330
add_text() (ezdxf.layouts.BaseLayout method), 157
add_to_layout() (ezdxf.math.ConstructionArc method), 285
add_to_layout() (ezdxf.math.ConstructionEllipse method), 287
add_trace() (ezdxf.layouts.BaseLayout method), 156
add_underlay() (ezdxf.layouts.BaseLayout method), 162
add_underlay_def() (ezdxf.drawing.Drawing method), 116
add_underlay_def() (ezdxf.sections.objects.ObjectsSection method), 123
add_vertices() (ezdxf.render.MeshBuilder method), 320
add_viewport() (ezdxf.layouts.Paperspace method), 171
add_xref_def() (ezdxf.drawing.Drawing method), 116
add_xref_def() (ezdxf.sections.objects.ObjectsSection method), 123
adjust_for_background (ezdxf.entities.Underlay attribute), 239
align_angle (ezdxf.entities.ExtrudedSurface.dxf attribute), 233
align_angle (ezdxf.entities.SweptSurface.dxf at-
base_point_set (ezdxf.entities.RevolvedSurface.dxf attribute), 234

axis_rotate () (ezdxf.math.Matrix44 class method), 272

axis_vector (ezdxf.entities.RevolvedSurface.dxf attribute), 234

back_clip_plane_z_value (ezdxf.entities.Viewport.dxf attribute), 241

back_clipping (ezdxf.entities.View.dxf attribute), 140

back_clipping (ezdxf.entities.VPort.dxf attribute), 139

background_handle (ezdxf.entities.View.dxf attribute), 141

background_handle (ezdxf.entities.Viewport.dxf attribute), 243

banded_matrix () (in module ezdxf.math), 297

BandedMatrixLU (class in ezdxf.math), 300

bank (ezdxf.entities.ExtrudedSurface.dxf attribute), 233

bank (ezdxf.entities.SweptSurface.dxf attribute), 235

base_point (ezdxf.entities.Block.dxf attribute), 145

base_point (ezdxf.entities.PatternLine attribute), 205

base_point_set (ezdxf.entities.ExtrudedSurface.dxf attribute), 233

base_point_set (ezdxf.entities.SweptSurface.dxf attribute), 235

BaseLayout (class in ezdxf.layouts), 153

Bezier (class in ezdxf.math), 293

Bezier (class in ezdxf.render), 312

Bezier4P (class in ezdxf.math), 293

bezier_decomposition () (ezdxf.math.BSpline method), 292

BezierSurface (class in ezdxf.math), 294

bg_fill (ezdxf.entities.MText.dxf attribute), 216

bg_fill_color (ezdxf.entities.MText.dxf attribute), 216

bg_fill_color_name (ezdxf.entities.MText.dxf attribute), 216

bg_fill_true_color (ezdxf.entities.MText.dxf attribute), 216

bgcolor (ezdxf.entities.Hatch attribute), 197

bigfont (ezdxf.entities.Textstyle.dxf attribute), 130

bisectrix () (ezdxf.math.ConstructionRay method), 281

blend_collapse (ezdxf.entities.Mesh.dxf attribute), 214

Block (class in ezdxf.entities), 144

block (ezdxf.layouts.BlockLayout attribute), 172
Index

443
defpoint3 (ezdxf.entities.Dimension.dxf attribute), 187
defpoint4 (ezdxf.entities.Dimension.dxf attribute), 187
defpoint5 (ezdxf.entities.Dimension.dxf attribute), 187
degree (ezdxf.entities.Spline.dxf attribute), 230
degree (ezdxf.entities.SplineEdge attribute), 204
degree (ezdxf.math.BSpline attribute), 290
del_dxf_attrib() (ezdxf.entities.DXFEntity method), 176
delete() (ezdxf.entities.dxgroups.GroupCollection method), 174
delete() (ezdxflayouts.Layouts method), 152
delete_all_attribs() (ezdxf.entities.Insert method), 149
delete_all_blocks() (ezdxf.sections.blocks.BlocksSection method), 121
delete_all_entities() (ezdxf.layouts.BaseLayout method), 154
delete_attrib() (ezdxf.entities.Insert method), 148
delete_block() (ezdxf.sections.blocks.BlocksSection method), 120
delete_config() (ezdxf.sections.table.ViewportTable method), 126
delete_entity() (ezdxf.entitydb.EntityDB method), 424
delete_entity() (ezdxf.layouts.BaseLayout method), 154
delete_layout() (ezdxf.drawing.Drawing method), 115
derivative() (ezdxf.math.Bezier method), 293
derivative() (ezdxf.math.BSpline method), 291
derivatives() (ezdxf.math.Bezier method), 293
derivatives() (ezdxf.math.BSpline method), 291
description (ezdxf.addons.acadctb.ColorDependentPlotStyles attribute), 340
description (ezdxf.addons.acadctb.NamedPlotStyles attribute), 341
description (ezdxf.addons.acadctb.PlotStyle attribute), 342
description (ezdxf.entities.dxgroups.DXFGroup.dxf attribute), 173
description (ezdxf.entities.Layer attribute), 128
description (ezdxf.entities.Linetype.dxf attribute), 130
design_point (ezdxf.entities.GeoData.dxf attribute), 248
detect_banded_matrix() (in module ezdxf.math), 297
determinant() (ezdxf.math.BandedMatrixLU method), 301
determinant() (ezdxf.math.LUDecomposition method), 300
determinant() (ezdxf.math.Matrix method), 299
determinant() (ezdxf.math.Matrix44 method), 273
dgnDefinition (class in ezdxf.entities), 253
dgnUnderlay (class in ezdxf.entities), 240
diag() (ezdxf.math.Matrix method), 298
dictionary (class in ezdxf.entities), 245
dictionaryWithDefault (class in ezdxf.entities), 247
dimadec (ezdxf.entities.DimStyle.dxf attribute), 133
dimalt (ezdxf.entities.DimStyle.dxf attribute), 133
dimaltd (ezdxf.entities.DimStyle.dxf attribute), 133
dimaltf (ezdxf.entities.DimStyle.dxf attribute), 132
dimaltnd (ezdxf.entities.DimStyle.dxf attribute), 132
dimalttdd (ezdxf.entities.DimStyle.dxf attribute), 133
dimaltttz (ezdxf.entities.DimStyle.dxf attribute), 134
dimaltu (ezdxf.entities.DimStyle.dxf attribute), 133
dimaltz (ezdxf.entities.DimStyle.dxf attribute), 134
dimapost (ezdxf.entities.DimStyle.dxf attribute), 131
dimasz (ezdxf.entities.DimStyle.dxf attribute), 131
dimatfit (ezdxf.entities.DimStyle.dxf attribute), 135
dimunit (ezdxf.entities.DimStyle.dxf attribute), 134
dimazin (ezdxf.entities.DimStyle.dxf attribute), 133
dimblk (ezdxf.entities.DimStyle.dxf attribute), 131
dimblkl1 (ezdxf.entities.DimStyle.dxf attribute), 131
dimblkl1_handle (ezdxf.entities.DimStyle.dxf attribute), 135
dimblkl2 (ezdxf.entities.DimStyle.dxf attribute), 131
dimblkl2_handle (ezdxf.entities.DimStyle.dxf attribute), 135
dimblk_handle (ezdxf.entities.DimStyle.dxf attribute), 135
dimc1rd (ezdxf.entities.DimStyle.dxf attribute), 131
dimc1re (ezdxf.entities.DimStyle.dxf attribute), 133
dimc1rt (ezdxf.entities.DimStyle.dxf attribute), 133
dimStylesoc (ezdxf.entities.DimStyle.dxf attribute), 133
dimdle (ezdxf.entities.DimStyle.dxf attribute), 131
dimdli (ezdxf.entities.DimStyle.dxf attribute), 131
dimdsep (ezdxf.entities.DimStyle.dxf attribute), 131
Dimension (class in ezdxf.entities), 186
dimension (ezdxf.entities.DimStyleOverride attribute), 189
dimelex (ezdxf.entities.DimStyle.dxf attribute), 131
dimeixo (ezdxf.entities.DimStyle.dxf attribute), 131
dimfit (ezdxf.entities.DimStyle.dxf attribute), 134
dimfrac (ezdxf.entities.DimStyle.dxf attribute), 134
dimfx1 (ezdxf.entities.DimStyle.dxf attribute), 135
dimfxlon (ezdxf.entities.DimStyle.dxf attribute), 135
dimgap (ezdxf.entities.DimStyle.dxf attribute), 132
dimjust (ezdxf.entities.DimStyle.dxf attribute), 134
dimlbbrlk (ezdxf.entities.DimStyle.dxf attribute), 135
dimlbbrlk_handle (ezdxf.entities.DimStyle.dxf attribute), 135
edge_create_values (ezdxf.entities.MeshData attribute), 214
edge_data (ezdxf.entities.MeshData attribute), 214
edges (ezdxf.entities.MeshData attribute), 214
edges (ezdxf.entities.MeshBuilder attribute), 320
edges_as_vertices () (ezdxf.entities.MeshBuilder method), 320
edit_data () (ezdxf.entities.Mesh method), 214
elevation (ezdxf.entities.Hatch.dxf attribute), 197
elevation (ezdxf.entities.LWPolyline.dxf attribute), 211
elevation (ezdxf.entities.Polyline.dxf attribute), 221
elevation (ezdxf.entities.View.dxf attribute), 141
elevation (ezdxf.entities.Viewport.dxf attribute), 243
Ellipse (class in ezdxf.entities), 194
eb () (in module ezdxf.render.forms), 315
ellipse_approximation () (ezdxf.math.BSpline static method), 292
ellipse_edges_to_spline_edges ()
(ezdxf.entities.BoundaryPaths method), 200
EllipsEdge (class in ezdxf.entities), 203
embedded_objects (ezdxf.lldxf.extendedtags.ExtendedTags attribute), 429
encode () (in module ezdxf.tools.crypt), 303
encode_base64 () (ezdxf.drawing.Drawing method), 114
encoding (ezdxf.drawing.Drawing attribute), 111
end (ezdxf.entities.Line.dxf attribute), 209
end (ezdxf.entities.LineEdge attribute), 203
end (ezdxf.math.ConstructionEllipse attribute), 286
end (ezdxf.math.ConstructionLine attribute), 281
end_angle (ezdxf.entities.Arc.dxf attribute), 183
end_angle (ezdxf.entities.ArcDimension.dxf attribute), 193
end_angle (ezdxf.entities.ArcEdge attribute), 203
end_angle (ezdxf.entities.EllipseEdge attribute), 203
end_angle (ezdxf.math.ConstructionArc attribute), 284
end_angle_rad (ezdxf.math.ConstructionArc attribute), 284
end_draft_angle (ezdxf.entities.LoftedSurface.dxf attribute), 233
end_draft_distance
(ezdxf.entities.RevolvedSurface.dxf attribute), 234
end_draft_magnitude
(ezdxf.entities.LoftedSurface.dxf attribute), 233
end_param (ezdxf.entities.Ellipse.dxf attribute), 194
end_point (ezdxf.entities.Arc attribute), 183
end_point (ezdxf.entities.Ellipse attribute), 194
end_point (ezdxf.math.ConstructionArc attribute), 284
end_point (ezdxf.math.ConstructionEllipse attribute), 286
end_style (ezdxf.addons.acadctb.PlotStyle attribute), 343
end_tangent (ezdxf.entities.Spline.dxf attribute), 230
end_tangent (ezdxf.entities.SplineEdge attribute), 204
end_width (ezdxf.entities.Vertex.dxf attribute), 224
finalize() (ezdxf.addons.importer.Importer method), 334
find_all() (ezdxf.ldxf.tags.Tags method), 428
find_shx() (ezdxf.sections.table.StyleTable method), 125
first (ezdxf.query.EntityQuery attribute), 255
fit_point_count() (ezdxf.entities.Spline method), 231
fit_points (ezdxf.entities.Spline method), 231
fit_points (ezdxf.entities.SplineEdge attribute), 204
fit_points_to_cad_cv() (in module ezdxf.math), 264
fit_tolerance (ezdxf.entities.Spline.dxf attribute), 230
flag (guide.ExampleCls attribute), 433
flags (ezdxf.entities.AppID.dxf attribute), 142
flags (ezdxf.entities.Block.dxf attribute), 145
flags (ezdxf.entities.Body.dxf attribute), 184
flags (ezdxf.entities.Dimension.dxf attribute), 131
flags (ezdxf.entities.DXFClass.dxf attribute), 118
flags (ezdxf.entities.Image.dxf attribute), 206
flags (ezdxf.entities.Layer.dxf attribute), 126
flags (ezdxf.entities.LWPolyline.dxf attribute), 211
flags (ezdxf.entities.Polyline.dxf attribute), 221
flags (ezdxf.entities.Spline.dxf attribute), 230
flags (ezdxf.entities.Textstyle.dxf attribute), 129
flags (ezdxf.entities.UCSTable.dxf attribute), 142
flags (ezdxf.entities.Underlay.dxf attribute), 239
flags (ezdxf.entities.Vertex.dxf attribute), 225
flags (ezdxf.entities.View.dxf attribute), 140
flags (ezdxf.entities.Viewport.dxf attribute), 242
flags (ezdxf.entities.VPort.dxf attribute), 138
flatten_subclasses() (ezdxf.ldxf.extendedtags.ExtendedTags method), 430
float2transparency() (in module ezdxf.tools), 302
flow_direction (ezdxf.entities.MText.dxf attribute), 215
font (ezdxf.entities.Textstyle.dxf attribute), 130
freeze() (ezdxf.entities.Layer method), 128
freeze() (ezdxf.math.Matrix method), 299
freeze_matrix() (in module ezdxf.math), 298
from_2p_angle() (ezdxf.math.ConstructionArc class method), 284
from_2p_radius() (ezdxf.math.ConstructionArc class method), 285
from_3p() (ezdxf.math.ConstructionArc class method), 285
from_3p() (ezdxf.math.ConstructionCircle static method), 282
from_3p() (ezdxf.math.Plane class method), 278
from_angle() (ezdxf.math.Vector class method), 276
from_arc() (ezdxf.entities.Ellipse class method), 195
from_arc() (ezdxf.entities.Spline class method), 232
from_arc() (ezdxf.math.BSpline static method), 292
from_arc() (ezdxf.math.ConstructionEllipse class method), 287
from_builder() (ezdxf.render.MeshBuilder class method), 322
from_deg_angle() (ezdxf.math.Vector class method), 276
from_ellipse() (ezdxf.math.BSpline static method), 292
from_file() (in module ezdxf.comments), 302
from_fit_points() (ezdxf.math.BSpline static method), 292
from_mesh() (ezdxf.render.MeshBuilder class method), 322
from_points() (ezdxf.math.ConstructionBox class method), 288
from_polyface() (ezdxf.render.MeshBuilder class method), 322
from_profiles_linear() (in module ezdxf.render.forms), 319
from_profiles_spline() (in module ezdxf.render.forms), 319
from_stream() (in module ezdxf.comments), 302
from_tags() (ezdxf.ldxf.packedtags.TagList class method), 431
from_tags() (ezdxf.ldxf.packedtags.VertexArray class method), 432
from_text() (ezdxf.ldxf.extendedtags.ExtendedTags class method), 431
from_text() (ezdxf.ldxf.tags.Tags class method), 427
from_vector() (ezdxf.math.Plane class method), 278
from_wcs() (ezdxf.math.OCS method), 267
from_wcs() (ezdxf.math.UCS method), 268
from_x_axis_and_point_in_xy() (ezdxf.math.UCS static method), 269
from_x_axis_and_point_in_xz() (ezdxf.math.UCS static method), 269
from_y_axis_and_point_in_xy() (ezdxf.math.UCS static method), 270
from_y_axis_and_point_in_yz() (ezdxf.math.UCS static method), 270
from_z_axis_and_point_in_xz() (ezdxf.math.UCS static method), 270
from_z_axis_and_point_in_yz() (ezdxf.math.UCS static method), 270
front_clip_plane_z_value (ezdxf.entities.Viewport.dxf attribute), 241
front_clipping (ezdxf.entities.View.dxf attribute), 140
front_clipping (ezdxf.entities.VPort.dxf attribute), 139
frozen_layers (ezdxf.entities.Viewport attribute), 244
G

gauss_jordan_inverse() (in module ezdxf.math), 295
gauss_jordan_solver() (in module ezdxf.math), 295
gauss_matrix_solver() (in module ezdxf.math), 296
gauss_vector_solver() (in module ezdxf.math), 296
gear() (in module ezdxf.render.forms), 317
generate() (ezdxf.math.Vector class method), 275
generation_flags (ezdxf.entities.Textstyle.dxf attribute), 129
geo_rss_tag (ezdxf.entities.GeoData.dxf attribute), 249
GeoData (class in ezdxf.entities), 247
geometry (ezdxf.entities.Dimension.dxf attribute), 186
get() (ezdxf.entities.Dictionary), 246
get() (ezdxf.entities.DictionaryWithDefault), 247
get() (ezdxf.entities.DimStyleOverride method), 189
get() (ezdxf.entities.dxfformats.GroupCollection method), 174
get() (ezdxf.entitydb.EntityDB method), 424
get() (ezdxf.layouts.Layouts method), 152
get() (ezdxf.sections.blocks.BlocksSection method), 120
get() (ezdxf.sections.classes.ClassesSection method), 118
get() (ezdxf.sections.header.HeaderSection method), 117
get() (ezdxf.sections.header.HeaderSection method), 117
get() (ezdxf.sections.table.Table method), 124
get_align() (ezdxf.entities.Text method), 237
get_app_data() (ezdxf.entities.DXFEntity method), 176
get_app_data() (ezdxf.ildxf.extendedtags.ExtendedTags method), 430
get_app_data_content() (ezdxf.ildxf.extendedtags.ExtendedTags method), 430
get_arrow_names() (ezdxf.entities.DimStyleOverride method), 190
get_attdef() (ezdxf.layouts.BlockLayout method), 172
get_attdef_text() (ezdxf.layouts.BlockLayout method), 172
get_attrib() (ezdxf.entities.Insert method), 148
get_attrib_text() (ezdxf.entities.Insert method), 148
get_col() (ezdxf.math.Matrix44 method), 271
get_color() (ezdxf.entities.Layer method), 128
get_config() (ezdxf.sections.table.ViewportTable method), 126
get_dim_style() (ezdxf.entities.Dimension method), 188
get_dxf_attrib() (ezdxf.entities.DXFEntity method), 175
get_extension_dict() (ezdxf.entities.DXFEntity method), 176
get_extension_dict() (ezdxf.layouts.BaseLayout method), 154
get_first_tag() (ezdxf.ildxf.tags.Tags method), 428
get_first_value() (ezdxf.ildxf.tags.Tags method), 428
get_flag_state() (ezdxf.entities.DXFEntity method), 176
get_geodata() (ezdxf.layouts.Modelspace method), 170
get_geometry_block() (ezdxf.entities.DimStyleOverride method), 170
get_handle() (ezdxf.ildxf.extendedtags.ExtendedTags method), 429
get_handle() (ezdxf.ildxf.tags.Tags method), 427
get_hyperlink() (ezdxf.entities.DXFGraphic method), 179
get_image_def() (ezdxf.entities.Image method), 207
get_layout() (ezdxf.entities.DXFGraphic method), 178
get_layout_for_entity() (ezdxf.layouts.Layouts method), 153
get_lineweight() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
get_lineweight() (ezdxf.addons.acadctb.NamedPlotStyles method), 341
get_lineweight_index() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
get_lineweight_index() (ezdxf.addons.acadctb.NamedPlotStyles method), 342
get_measurement() (ezdxf.entities.Dimension method), 188
get_mesh_vertex() (ezdxf.entities.Polymesh method), 225
get_mesh_vertex_cache() (ezdxf.entities.Polymesh method), 226
get_mode() (ezdxf.entities.Polylne method), 223
get_name() (ezdxf.entities.dxfformats.DXFGroup method), 173
get_paper_limits() (ezdxf.layouts.Paperspace method), 171
get_points() (ezdxf.entities.LWPolyline method), 213
get_pos() (ezdxf.entities.Text method), 237

Index
get_reactors() (ezdxf.entities.DXFEntity method), 177
get_redraw_order() (ezdxf.layouts.Layout method), 170
get_required_dict() (ezdxf.entities.Dictionary method), 246
groupby() (ezdxf.groupby method), 127
groupby() (ezdxf.groupby method), 177
groups() (ezdxf.entities.Dictionary method), 246
groups() (ezdxf.entities.dxfgroups.GroupCollection method), 174
groupby() (ezdxf.math.Matrix44 method), 271
groupby() (ezdxf.query.EntityQuery method), 255
grid_spacing (ezdxf.entities.DXFEntity method), 170
grid_spacing (ezdxf.math.Matrix44 method), 271
grid_on() (ezdxf.math.Matrix44 method), 271
get_xdata() (ezdxfentities.dxfgroups.GroupCollection method), 174
get_subclass() (ezdxf.lldxf.extendedtags.ExtendedTags method), 430
get_table_lineweight() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
get_table_lineweight() (ezdxf.addons.acadctb.NamedPlotStyles method), 342
get_xdata() (ezdxf.entities.DXFEntity method), 176
get_xdata() (ezdxf.lldxf.extendedtags.ExtendedTags method), 430
get_xdata_list() (ezdxf.entities.DXFEntity method), 177
global_bspline_interpolation() (in module ezdxf.math), 265
Gradient (class in ezdxf.entities), 205
graphic_properties() (ezdxf.entities.DXFFGraphic method), 179
grayscale (ezdxf.addons.acadctb.PlotStyle attribute), 343
grid() (ezdxf.entities.Insert method), 147
grid_frequency (ezdxf.entities.Viewport.dxf attribute), 243
grid_on() (ezdxf.entities.VPort.dxf attribute), 139
grid_spacing (ezdxf.entities.Viewport.dxf attribute), 241
grid_spacing (ezdxf.entities.VPort.dxf attribute), 139
group_tags() (in module ezdxf.lldxf.tags), 429
groupby() (ezdxf.lldxf.tags.Tag method), 429
groupby() (ezdxf.math.Matrix44 method), 271
groupby() (ezdxf.query.EntityQuery method), 255
groupby() (in module ezdxf.groupby), 256
GroupCollection (class in ezdxf.entities.dxfgroups), 174
groups() (ezdxf.lldxf.drawing.Drawing attribute), 112
groups() (ezdxf.entities.dxfgroups.GroupCollection method), 174
guid() (in module ezdxf.tools), 302
guide (module), 433

H
halign (ezdxf.entities.Text.dxf attribute), 236
handle (ezdxf.entities.DXFEntity.dxf attribute), 175
handle (ezdxf.entities.EndBlk.dxf attribute), 146
handle (ezdxf.entities.Layer.dxf attribute), 126
handle (ezdxf.entities.Textstyle.dxf attribute), 129
handles() (ezdxf.entities.dxfgroups.DXFGroup method), 173
hardOwned (ezdxf.entities.Dictionary.dxf attribute), 245
has_app_data() (ezdxf.entities.DXFEntity method), 176
has_app_data() (ezdxf.lldxf.extendedtags.ExtendedTags method), 430
has_arc (ezdxf.entities.LWPolyline attribute), 211
has_arc (ezdxf.entities.Polylines attribute), 223
has_arrowhead (ezdxf.entities.Leader.dxf attribute), 207
has_attdef() (ezdxf.layouts.BlockLayout method), 172
has_attrib() (ezdxf.entities.Insert method), 147
has_binary_data (ezdxf.entities.Body attribute), 184
has_data (ezdxf.math.BoundingBox attribute), 279
has_data (ezdxf.math.BoundingBox2d attribute), 280
has_dxf_attrib() (ezdxf.entities.DXFEntity method), 175
has_embedded_objects() (ezdxf.lldxf.tags.Tags method), 428
has_entry() (ezdxf.sections.table.Table method), 124
has_extension_dict() (ezdxf.entities.DXFEntity method), 176
has_gradient_data (ezdxf.entities.Hatch attribute), 197
has_handle() (ezdxf.entitydb.EntitySpace method), 425
has_hookline (ezdxf.entities.Leader.dxf attribute), 208
has_hyperlink() (ezdxf.entities.DXFFGraphic method), 179
has_intersection() (ezdxf.math.ConstructionLine method), 282
has_leader (ezdxf.entities.ArcDimension.dxf attribute), 193
has_line() (ezdxf.math.BoundingBox2d method), 321
has_pattern_fill (ezdxf.entities.Hatch attribute), 197
has_reactors() (ezdxf.entities.DXFEntity method), 177
has_scaling (ezdxf.entities.Insert attribute), 147
has_solid_fill (ezdxf.entities.Hatch attribute), 197
has_tag() (ezdxf.lldxf.tags.Tags method), 428
has_tag() (ezdxf.sections.header.CustomVars method), 118
has_uniform_scaling (ezdxf.entities.Insert attribute), 147

Index 451
iter_col() (ezdxf.math.Matrix method), 298
iter_diag() (ezdxf.math.Matrix method), 299
iter_row() (ezdxf.math.Matrix method), 298
IterDXF (class in ezdxf.addons.iterdxf), 332
IterDXFReader (class in ezdxf.addons.iterdxf), 332

J
join_style (ezdxf.addons.acadctb.PlotStyle attribute), 343
julian_day (ezdxf.entities.Sun.dxf attribute), 251
juliandate() (in module ezdxf.tools), 302

K
key (ezdxf.entities.DXFClass attribute), 119
key() (ezdxf.sections.table.Table static method), 124
keys() (ezdxf.entitydb.EntityDB method), 246
knot_count() (ezdxf.entities.Spline method), 231
knot_tolerance (ezdxf.entities.Spline method), 230
knot_values (ezdxf.entities.SplineEdge attribute), 204
knots (ezdxf.entities.Spline attribute), 231
knots() (ezdxf.math.BSpline method), 291

L
last (ezdxf.query.EntityQuery attribute), 255
last_height (ezdxf.entities.Textstyle.dxf attribute), 129
Layer (class in ezdxf.entities), 126
layer (ezdxf.entities.Block.dxf attribute), 145
layer (ezdxf.entities.DXFGraphic.dxf attribute), 180
layer (ezdxf.entities.EndBlk.dxf attribute), 146
layers (ezdxf.addons.dxf2code.Code attribute), 339
layers (ezdxf.drawing.Drawing attribute), 113
layers (ezdxf.sections.tables.TablesSection attribute), 119
LayerTable (class in ezdxf.sections.table), 124
Layout (class in ezdxf.layouts), 168
layout (ezdxf.entities.BlockRecord.dxf attribute), 143
layout() (ezdxf.drawing.Drawing method), 114
layout_names() (ezdxf.drawing.Drawing method), 114
layout_names_in_table() (ezdxf.drawing.Drawing method), 114
Layouts (class in ezdxf.layouts), 152
layouts (ezdxf.drawing.Drawing attribute), 112
layouts_and_blocks() (ezdxf.drawing.Drawing attribute), 116
Leader (class in ezdxf.entities), 207
leader_length (ezdxf.entities.Dimension.dxf attribute), 187
leader_offset_annotation_placement (ezdxf.entities.Leader.dxf attribute), 208
leader_offset_block_ref (ezdxf.entities.Leader.dxf attribute), 208
leader_point1 (ezdxf.entities.ArcDimension.dxf attribute), 193
leader_point2 (ezdxf.entities.ArcDimension.dxf attribute), 193
legacy_repair() (ezdxf.lldxf.extendedtags.ExtendedTags method), 430
length (ezdxf.entities.Line.dxf attribute), 130
length() (ezdxf.math.ConstructionLine method), 281
length (ezdxf.entities.View.dxf attribute), 140
length (ezdxf.entities.VPort.dxf attribute), 139
lerp() (ezdxf.math.Vector method), 276
Line (class in ezdxf.entities), 209
line_spacing_factor (ezdxf.entities.Dimension.dxf attribute), 188
line_spacing_factor (ezdxf.entities.MText.dxf attribute), 216
line_spacing_style (ezdxf.entities.Dimension.dxf attribute), 187
line_spacing_style (ezdxf.entities.MText.dxf attribute), 216
LineEdge (class in ezdxf.entities), 203
linepattern_size (ezdxF.addons.acadctb.PlotStyle attribute), 342
lines (ezdxF.entities.Pattern attribute), 204
Linetype (class in ezdxF.entities), 130
linetype (ezdxF.addons.acadctb.PlotStyle attribute), 342
linetype (ezdxF.entities.DXFGraphic.dxf attribute), 180
linetype (ezdxF.entities.Layer.dxf attribute), 127
linetypes (ezdxF.addons.dxf2code.Code attribute), 339
linetypes (ezdxF.drawing.Drawing attribute), 113
linetypes (ezdxF.sections.tables.TablesSection attribute), 119
lineweight (ezdxF.addons.acadctb.PlotStyle attribute), 342
lineweight (ezdxF.entities.DXFGraphic.dxf attribute), 180
lineweight (ezdxF.entities.Layer.dxf attribute), 127
lineweights (ezdxF.addons.acadctb.ColorDependentPlotStyles attribute), 340
lineweights (ezdxF.addons.acadctb.NamedPlotStyles attribute), 341
linspace() (in module ezdxF.math), 258
list() (ezdxF.math.Vector class method), 276
live_selection_handle (ezdxF.entities.View.dxf attribute), 141
load() (in module ezdxF.addons.acadctb), 339
load_proxy_graphics (in module ezdxF.options), 301
loaded (ezdxF.entities.ImageDef.dxf attribute), 249
Index
points_from_wcs() (ezdxf.math.OCS method), 267
points_from_wcs() (ezdxf.math.UCS method), 268
points_to_ocs() (ezdxf.math.OCS method), 268
points_to_wcs() (ezdxf.math.OCS method), 267
points_to_ucs() (ezdxf.math.UCS method), 268
Polyface (class in ezdxf.entities), 226
Polyline (class in ezdxf.entities), 221
polyline_to_edge_path() (ezdxf.entities.BoundaryPaths method), 200
PolylinePath (class in ezdxf.entities), 201
Polymesh (class in ezdxf.entities), 225
pop() (ezdxf.entities.DimStyleOverride method), 190
pop_tags() (ezdxf.ldxf.tags.Tags method), 429
preserve_proxy_graphics() (in module ezdxf.options), 301
prev_plot_init() (ezdxf.layouts.Layout method), 170
print_lineweights() (ezdxf.layouts.Layout method), 170
priority() (ezdxf.entities.DXFEntity attribute), 175
project() (ezdxf.math.Vector method), 276
prompt (ezdxf.entities.Attdef.dxf attribute), 151
properties (ezdxf.sections.header.CustomVars attribute), 117
purge() (ezdxf.entitydb.EntitySpace method), 425
pyramids() (ezdxf.addons.SierpinskyPyramid method), 359
query() (ezdxf.drawing.Drawing method), 114
query() (ezdxf.layouts.BaseLayout method), 154
query() (ezdxf.query.EntityQuery method), 255
query() (ezdxf.sections.objects.ObjectsSection method), 122

R
R12FastStreamWriter (class in ezdxf.addons.r12writer), 326
R12writer (class in ezdxf.render), 312
r12writer() (in module ezdxf.addons.r12writer), 326
radius (ezdxf.entities.Arc.dxf attribute), 203
radius (ezdxf.entities.Circle.dxf attribute), 185
radius (ezdxf.entities.Ellipse.dxf attribute), 203
radius (ezdxf.math.ConstructionArc attribute), 284
radius (ezdxf.math.ConstructionCircle attribute), 282
radius() (ezdxf.math.EulerSpiral method), 295
random_2d_path() (in module ezdxf.render), 314
random_3d_path() (in module ezdxf.render), 314
ratio (ezdxf.entities.Ellipse.dxf attribute), 194
ratio (ezdxf.math.ConstructionElliptical attribute), 286
rational (ezdxf.entities.SplineEdge attribute), 204
rational_spline_from_arc() (in module ezdxf.math), 266
rational_spline_from_ellipse() (in module ezdxf.math), 266
Ray (class in ezdxf.entities), 227
ray (ezdxf.math.ConstructionLine attribute), 281
read() (in module ezdxf), 110
readfile() (in module ezdxf), 109
readzip() (in module ezdxf), 110
recreate_source_layout() (ezdxf.addons.importer.Importer method), 336
ref_vp_object_1 (ezdxf.entities.Viewport.dxf attribute), 244
ref_vp_object_2 (ezdxf.entities.Viewport.dxf attribute), 244
ref_vp_object_3 (ezdxf.entities.Viewport.dxf attribute), 244
ref_vp_object_4 (ezdxf.entities.Viewport.dxf attribute), 244
reference_point (ezdxf.entities.GeoData.dxf attribute), 248
reference_vector_for_controlling_twist (ezdxf.entities.ExtrudedSurface.dxf attribute), 233
reference_vector_for_controlling_twist (ezdxf.entities.SweptSurface.dxf attribute), 235
Region (class in ezdxf.entities), 228
register() (ezdxf.sections.classes.ClassesSection method), 118
remove() (ezdxf.entities.Dictionary method), 246
remove() (ezdxf.entitydb.EntitySpace method), 425
remove() (ezdxf.query.EntityQuery method), 255
remove() (ezdxf.sections.header.CustomVars method), 118
remove() (ezdxf.sections.table.Table method), 124
remove_association() (ezdxf.entities.Hatch method), 200
remove_tags() (ezdxf.ldrxf.tags.Tags method), 429
remove_tags_except() (ezdxf.ldrxf.tags.Tags method), 429
rename() (ezdxf.entities.Layer method), 128
rename() (ezdxf.layouts.Layouts method), 152
rename() (ezdxf.layouts.Paperspace method), 171
rename_block() (ezdxf.sections.blocks.BlocksSection method), 120
render() (ezdxf.addons.MengerSponge method), 354
render() (ezdxf.addons.SierpinskyPyramid method), 359
render() (ezdxf.entities.Dimension method), 188
render() (ezdxf.entities.DimStyleOverride method), 193
render() (ezdxf.render.Bezier method), 313
render() (ezdxf.render.MeshBuilder method), 321
render() (ezdxf.render.R12Spline method), 312
render_3dfaces() (ezdxf.render.MeshBuilder method), 321
remove_tags() (ezdxf.ldrxf.tags.Tags method), 429
remove_tags_except() (ezdxf.ldrxf.tags.Tags method), 429
rename() (ezdxf.entities.Layer method), 128
rename() (ezdxf.layouts.Layouts method), 152
rename() (ezdxf.layouts.Paperspace method), 171
rename_block() (ezdxf.sections.blocks.BlocksSection method), 120
render() (ezdxf.addons.MengerSponge method), 354
render() (ezdxf.addons.SierpinskyPyramid method), 359
render() (ezdxf.entities.Dimension method), 188
render() (ezdxf.entities.DimStyleOverride method), 193
render() (ezdxf.render.Bezier method), 313
render() (ezdxf.render.MeshBuilder method), 321
render() (ezdxf.render.R12Spline method), 312
render_3dfaces() (ezdxf.render.MeshBuilder method), 321
method), 321
render_as_fit_points() (ezdxf.render.Spline
method), 310
render_axis() (ezdxf.math.OCS method), 267
render_axis() (ezdxf.math.UCS method), 270
render_closed_bspline() (ezdxf.render.Spline
method), 311
render_closed_rbspline() (ezdxf.render.Spline
method), 311
render_mode (ezdxf.entities.View.dxf attribute), 141
render_mode (ezdxf.entities.Viewport.dxf attribute),
242
render_normals() (ezdxf.render.MeshBuilder
method), 322
render_open_bspline() (ezdxf.render.Spline
method), 310
render_open_rbspline() (ezdxf.render.Spline
method), 311
render_polyface() (ezdxf.render.MeshBuilder
method), 321
render_polyline() (ezdxf.render.EulerSpiral
method), 313
render_spline() (ezdxf.render.EulerSpiral method),
313
render_uniform_bspline() (ezdxf.render.Spline
method), 311
render_uniform_rbspline()
(ezdxf.render.Spline method), 311
reorder() (ezdxf.entitydb.EntitySpace method), 425
replace() (ezdxf.entitydb.EntitySpace method), 425
replace() (ezdxf.sections.header.CustomVars
method), 118
replace_handle() (ezdxflldxf.extendedtags.ExtendedTags
method), 429
replace_handle() (ezdxflldxf.tags.Tags method),
427
replace_xdata_list() (ezdxf.entities.DXFEntity
method), 177
required_knot_values() (in module ezdxf.math),
257
reset_boundary_path() (ezdxf.entities.Image
method), 207
reset_boundary_path() (ezdxf.entities.Underlay
method), 240
reset_extends() (ezdxf.layouts.Layout method),
169
reset_fingerprint_guid() (ezdxf.drawing.Drawing method), 116
reset_paper_limits() (ezdxf.layouts.Paperspace
method), 171
reset_transformation() (ezdxf.entities.Insert
method), 149
reset_version_guid() (ezdxf.drawing.Drawing
method), 116
reset_viewports() (ezdxf.layouts.Paperspace
method), 171
reshape() (ezdxf.math.Matrix static method), 298
resolution_units (ezdxf.entities.ImageDef.dxf
attribute), 49
reverse() (ezdxf.math.BSpline method), 291
reversed() (ezdxf.math.Vector method), 276
revolve_angle (ezdxf.entities.RevolvedSurface.dxf
attribute), 234
RevolvedSurface (class in ezdxf.entities), 234
rgb (ezdxf.entities.DXFGraphic attribute), 178
rgb (ezdxf.entities.Layer attribute), 127
rgb2int() (in module ezdxf.tools), 303
rootdict (ezdxf.drawing.Drawing attribute), 112
rootdict (ezdxf.sections.objects.ObjectsSection
attribute), 121
rotate() (ezdxf.math.ConstructionBox method), 289
rotate() (ezdxf.math.Shape2d method), 290
rotate() (ezdxf.math.UCS method), 269
rotate() (ezdxf.math.Vector method), 277
rotate_axis() (ezdxf.entities.DXFGraphic method),
180
rotate_axis() (ezdxf.render.MeshTransformer
method), 323
rotate_deg() (ezdxf.math.Vector method), 278
rotate_local_x() (ezdxf.math.UCS method), 269
rotate_local_y() (ezdxf.math.UCS method), 269
rotate_local_z() (ezdxf.math.UCS method), 269
rotate_rad() (ezdxf.math.Shape2d method), 290
rotate_x() (ezdxf.entities.DXFGraphic method), 179
rotate_x() (ezdxf.render.MeshTransformer method),
323
rotate_y() (ezdxf.render.MeshTransformer method),
180
rotate_y() (ezdxf.render.MeshTransformer method),
323
rotate_z() (ezdxf.entities.DXFGraphic method), 180
rotate_z() (ezdxf.math.ConstructionArc method),
284
rotate_z() (ezdxf.render.MeshTransformer method),
323
rotation (ezdxf.entities.Gradient attribute), 205
rotation (ezdxf.entities.Insert.dxf attribute), 147
rotation (ezdxf.entities.MText.dxf attribute), 216
rotation (ezdxf.entities.Shape.dxf attribute), 228
rotation (ezdxf.entities.Text.dxf attribute), 236
rotation (ezdxf.entities.Underlay.dxf attribute), 239
rotation_form() (in module ezdxf.render.forms),
319
row() (ezdxf.math.Matrix method), 298
row_count (ezdxf.entities.Insert.dxf attribute), 147
row_spacing (ezdxf.entities.Insert.dxf attribute), 147
rows() (ezdxf.math.Matrix method), 299
rows() (ezdxf.math.Matrix44 method), 273
ruled_surface (ezdxf.entities.LoftedSurface.dxf attribute), 234
rytz_axis_construction() (in module ezdxf.math), 261

S
save() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 341
save() (ezdxf.addons.acadctb.NamedPlotStyles method), 342
save() (ezdxf.drawing.Drawing method), 113
saveas() (ezdxf.drawing.Drawing method), 113
scale (ezdxf.entities.BlockRecord.dxf attribute), 143
scale (ezdxf.entities.Layer.dxf attribute), 239
scale() (ezdxf.entities.DXFGraphic method), 179
scale() (ezdxf.entities.Pattern method), 204
scale() (ezdxf.math.ConstructionBox method), 289
scale() (ezdxf.math.Matrix44 class method), 271
scale() (ezdxf.math.Shape2d method), 290
scale_estimation_method (ezdxf.entities.GeoData.dxf attribute), 248
scale_factor (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 340
scale_factor (ezdxf.addons.acadctb.NamedPlotStyles method), 341
scale_factor (ezdxf.entities.ExtrudedSurface.dxf attribute), 233
scale_factor (ezdxf.entities.SweptSurface.dxf attribute), 235
scale_lineweights() (ezdxf.layouts.Layout method), 170
scale_uniform() (ezdxf.entities.DXFGraphic method), 179
scale_uniform() (ezdxf.math.ConstructionArc method), 284
scale_uniform() (ezdxf.math.Shape2d method), 290
scale_uniform() (ezdxf.render.MeshTransformer method), 323
scale_uniformly (ezdxf.layouts.BlockLayout attribute), 172
scale_x (ezdxf.entities.Underlay.dxf attribute), 239
scale_y (ezdxf.entities.Underlay.dxf attribute), 239
scale_z (ezdxf.entities.Underlay.dxf attribute), 239
schema (ezdxf.entities.dxf attribute), 247
screen (ezdxf.addons.acadctb.PlotStyle attribute), 342
sea_level_correction (ezdxf.entities.GeoData.dxf attribute), 248
sea_level_elevation (ezdxf.entities.GeoData.dxf attribute), 248
seeds (ezdxf.entities.Hatch attribute), 197
selectable (ezdxf.entities.dxffields.DXFGroup.dxf attribute), 173
set() (ezdxf.llldxf.extendedtags.ExtendedTags method), 430
set_arrows() (ezdxf.entities.DimStyle method), 136
set_arrows() (ezdxf.entities.DimStyleOverride method), 190
set_axis() (guide.ExampleCls method), 433
set_bg_color() (ezdxf.entities.MText method), 217
set_boundary_path() (ezdxf.entities.Image method), 207
set_closed() (ezdxf.entities.Spline method), 231
set_closed_rational() (ezdxf.entities.Spline method), 231
set_color() (ezdxf.math.Matrix method), 299
set_color() (ezdxf.math.Matrix44 method), 271
set_col() (ezdxf.math.Shape2d method), 290
set_data() (ezdxf.entities.dxffields.DXFGroup method), 173
set_default() (ezdxf.entities.DictionaryWithDefault method), 247
set_diag() (ezdxf.math.Matrix method), 299
set_dimline_format() (ezdxf.entities.DimStyle method), 137
set_dimline_format() (ezdxf.entities.DimStyleOverride method), 191
set_dxf_attrib() (ezdxf.entities.DXFEntity method), 175
set_extline1() (ezdxf.entities.DimStyle method), 137
set_extline1() (ezdxf.entities.DimStyleOverride method), 192
set_extline2() (ezdxf.entities.DimStyle method), 137
set_extline2() (ezdxf.entities.DimStyleOverride method), 192
set_extline_format() (ezdxf.entities.DimStyleOverride method), 192
set_first() (ezdxf.llldxf.tags.Tags method), 429
set_flag_state() (ezdxf.entities.DXFEntity method), 176
set_flag_state() (in module ezdxf.tools), 302
set_font() (ezdxf.entities.MText method), 217
set_gradient() (ezdxf.entities.Hatch method), 199
set_hyperlink() (ezdxf.entities.DXFGraphic method), 179
set_limits() (ezdxf.entities.DimStyle method), 138
set_limits() (ezdxf.entities.DimStyleOverride method), 191
set_location() (ezdxf.entities.DimStyleOverride method), 192
set_location() (ezdxf.entities.MText method), 217
set_mesh_vertex() (ezdxf.entities.Polymesh method), 226
set_modelspace_vport() (ezdxf.drawing.Drawing method), 116
set_open_rational() (ezdxf.entities.Spline method), 231
set_open_uniform() (ezdxf.entities.Spline method), 231
set_pattern_angle() (ezdxf.entities.Hatch method), 198
set_pattern_definition() (ezdxf.entities.Hatch method), 197
set_pattern_fill() (ezdxf.entities.Hatch method), 198
set_pattern_scale() (ezdxf.entities.Hatch method), 198
set_plot_flags() (ezdxf.layouts.Layout method), 170
set_plot_style() (ezdxf.layouts.Layout method), 169
set_plot_type() (ezdxf.layouts.Layout method), 169
set_plot_window() (ezdxf.layouts.Layout method), 169
set_points() (ezdxf.entities.LWPolyline method), 213
set_pos() (ezdxf.entities.Text method), 237
set_raster_variables() (ezdxf.drawing.Drawing method), 115
set_raster_variables() (ezdxf.sections.objects.ObjectsSection method), 123
set_reactors() (ezdxf.entities.DXFEntity method), 177
set_redraw_order() (ezdxf.layouts.Layout method), 169
set_rotation() (ezdxf.entities.MText method), 217
set_row() (ezdxf.math.Matrix method), 299
set_row() (ezdxf.math.Matrix44 method), 271
set_scale() (ezdxf.entities.Insert method), 147
set_seed_points() (ezdxf.entities.Hatch method), 199
set_solid_fill() (ezdxf.entities.Hatch method), 198
set_table_lineweight() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 341
set_table_lineweight() (ezdxf.addons.acadctb.NamedPlotStyles method), 342
set_text() (ezdxf.entities.Body method), 184
set_text() (ezdxf.entities.DimStyleOverride method), 190
set_text() (ezdxf.entities.DimStyleOverride method), 191
set_text() (ezdxf.entities.DimStyleOverride method), 192
set_text_align() (ezdxf.entities.DimStyle method), 136
set_text_align() (ezdxf.entities.DimStyleOverride method), 190
set_text_format() (ezdxf.entities.DimStyle method), 136
set_text_format() (ezdxf.entities.DimStyleOverride method), 190
set_ticks() (ezdxf.entities.DimStyle method), 136
set_ticks() (ezdxf.entities.DimStyleOverride method), 190
set_tolerance() (ezdxf.entities.DimStyle method), 137
set_tolerance() (ezdxf.entities.DimStyleOverride method), 190
set_transformation_matrix_lofted_entity (ezdxf.entities.LoftedSurface attribute), 234
set_uniform() (ezdxf.entities.Spline method), 231
set_uniform_rational() (ezdxf.entities.Spline method), 231
set_values() (ezdxf.lldxf.packedtags.TagArray method), 431
set_vertices() (ezdxf.entities.Leader method), 209
set_vertices() (ezdxf.entities.PolylinePath method), 201
set_wipeout_variables() (ezdxf.drawing.Drawing method), 115
set_wipeout_variables() (ezdxf.sections.objects.ObjectsSection method), 124
set_xdata() (ezdxf.entities.DXFEntity method), 177
set_xdata() (ezdxf.lldxf.extendedtags.ExtendedTags method), 430
set_xdata_list() (ezdxf.entities.DXFEntity method), 177
shade_plot_handle (ezdxf.entities.Viewport.dxf attribute), 243
shade_plot_mode (ezdxf.entities.Viewport.dxf attribute), 243
shadow_map_size (ezdxf.entities.Sun.dxf attribute), 251
shadow_mode (ezdxf.entities.DXFGraphic.dxf attribute), 181
shadow_softness (ezdxf.entities.Sun.dxf attribute), 251
shadow_type (ezdxf.entities.Sun.dxf attribute), 251
shadows (ezdxf.entities.Sun.dxf attribute), 251
Shape (class in ezdxf.entities), 228
shape (ezdxf.math.Matrix attribute), 298
Shape2d (class in ezdxf.math), 289
shift() (ezdxf.math.UCS method), 269
shift_text() (ezdxf.entities.DimStyleOverride method), 192
show_plot_styles() (ezdxf.layouts.Layout method), 170
SierpinskyPyramid (class in ezdxf.addons), 358
signed_distance_to() (ezdxf.math.Distance method), 279
simple_surfaces (ezdxf.entities.SweptSurface.dxf attribute), 234
single_pass_modelspace() (in module ezdxf.addons.iterdxf), 332
size (ezdxf.entities.Shape.dxf attribute), 228
size (ezdxf.math.BoundingBox attribute), 279
size (ezdxf.math.BoundingBox2d attribute), 280
slope (ezdxf.math.ConstructionRay attribute), 280
smooth_type (ezdxf.entities.PolylinePath.dxf attribute), 222
snap_angle (ezdxf.entities.Viewport.dxf attribute), 241
snap_base (ezdxf.entities.VPort.dxf attribute), 139
snap_base_point (ezdxf.entities.Viewport.dxf attribute), 241
snap_isopair (ezdxf.entities.VPort.dxf attribute), 139
snap_on (ezdxf.entities.VPort.dxf attribute), 139
snap_rotation (ezdxf.entities.VPort.dxf attribute), 139
snap_spacing (ezdxf.entities.Viewport.dxf attribute), 241
snap_spacing (ezdxf.entities.VPort.dxf attribute), 139
snap_style (ezdxf.entities.VPort.dxf attribute), 139
Solid (class in ezdxf.entities), 229
solid (ezdxf.entities.ExtrudedSurface.dxf attribute), 233
solid (ezdxf.entities.LoftedSurface.dxf attribute), 234
solid (ezdxf.entities.RevolvedSurface.dxf attribute), 234
solid (ezdxf.entities.SweptSurface.dxf attribute), 235
Solid3d (class in ezdxf.entities), 182
solid_fill (ezdxf.entities.Hatch.dxf attribute), 196
solve_matrix() (ezdxf.math.BandedMatrixLU method), 301
solve_vector() (ezdxf.math.BandedMatrixLU method), 301
solve_vector() (ezdxf.math.LUDecomposition method), 300
source_boundary_objects (ezdxf.entities.EdgePath attribute), 202
source_boundary_objects (ezdxf.entities.PolylinePath attribute), 201
source_vertices (ezdxf.entities.GeoData attribute), 249
spatial_angle (ezdxf.math.Vector attribute), 275
spatial_angle_deg (ezdxf.math.Vector attribute), 275
sphere() (in module ezdxf.render.forms), 318
Spline (class in ezdxf.entities), 230
Spline (class in ezdxf.render), 310
spline_edges_to_line_edges() (ezdxf.entities.BoundaryPaths method), 200
SplineEdge (class in ezdxf.entities), 203
square() (in module ezdxf.render.forms), 315
star() (in module ezdxf.render.forms), 316
start (ezdxf.entities.Line.dxf attribute), 209
start (ezdxf.entities.LineEdge attribute), 203
start (ezdxf.entities.Ray.dxf attribute), 227
start (ezdxf.entities.XLine.dxf attribute), 244
start (ezdxf.math.ConstructionElliptical attribute), 286
start (ezdxf.math.ConstructionLine attribute), 281
start() (ezdxf.render.Bezier method), 312
start_angle (ezdxf.entities.Arc.dxf attribute), 183
start_angle (ezdxf.entities.ArcDimension.dxf attribute), 193
start_angle (ezdxf.entities.ArcEdge attribute), 203
start_angle (ezdxf.entities.EllipticalEdge attribute), 203
start_angle (ezdxf.entities.RevolvedSurface.dxf attribute), 234
start_angle (ezdxf.math.ConstructionArc attribute), 284
start_angle_rad (ezdxf.math.ConstructionArc attribute), 284
start_draft_angle (ezdxf.entities.LoftedSurface.dxf attribute), 233
start_draft_distance (ezdxf.entities.RevolvedSurface.dxf attribute), 234
start_draft_magnitude (ezdxf.entities.LoftedSurface.dxf attribute), 233
start_param (ezdxf.entities.Ellipse.dxf attribute), 194
start_point (ezdxf.entities.Arc.dxf attribute), 183
start_point (ezdxf.entities.Elliptical attribute), 194
start_point (ezdxf.math.ConstructionArc attribute), 284
start_point (ezdxf.math.ConstructionElliptical attribute), 286
start_tangent (ezdxf.entities.Spline.dxf attribute), 230
start_tangent (ezdxf.entities.SplineEdge.dxf attribute), 230
start_width (ezdxf.entities.Vertex.dxf attribute), 224
status (ezdxf.entities.Sun.dxf attribute), 251
status (ezdxf.entities.Viewport.dxf attribute), 241
status (ezdxf.entities.VPort.dxf attribute), 139
STB, 434
store_proxy_graphics (in module ezdxf.options), 301
strip() (ezdxf.lldxf.tags.Tags class method), 429
style (ezdxf.entities.MText.dxf attribute), 216
style (ezdxf.entities.Text.dxf attribute), 236
styles (ezdxfaddons.dxf2code.Code attribute), 339
styles (ezdxf.sections.tables.TablesSection attribute), 119
StyleTable (class in ezdxf.sections.table), 125
subclasses (ezdxf.lldxf.extendedtags.ExtendedTags attribute), 429
subdivide() (ezdxf.render.MeshTransformer method), 322
subdivide() (ezdxf.render.Spline method), 310
subdivide_face() (in module ezdxf.math), 263
subdivide_ngons() (in module ezdxf.math), 263
subdivision_levels (ezdxf.entities.Mesh.dxf attribute), 214
subtract() (ezdxfaddons.pycsg.CSG method), 353
Sun (class in ezdxf.entities), 251
sun_handle (ezdxf.entities.View.dxf attribute), 142
sun_handle (ezdxf.entities.Viewport.dxf attribute), 244
suppress_zeros() (in module ezdxf.tools), 302
Surface (class in ezdxf.entities), 232
swap_axis() (ezdxf.math.ConstructionEllipse method), 287
swap_cols() (ezdxf.math.Matrix method), 299
swap_rows() (ezdxf.math.Matrix method), 299
sweep_alignment (ezdxf.entities.SweptSurface.dxf attribute), 235
sweep_alignment_flags (ezdxf.entities.ExtrudedSurface.dxf attribute), 233
sweep_entity_transform_computed (ezdxf.entities.ExtrudedSurface.dxf attribute), 233
sweep_entity_transform_computed (ezdxf.entities.SweptSurface.dxf attribute), 235
sweep_entity_transformation_matrix (ezdxf.entities.ExtrudedSurface attribute), 233
sweep_entity_transformation_matrix() (ezdxf.entities.SweptSurface method), 235
sweep_vector (ezdxf.entities.ExtrudedSurface.dxf attribute), 232
sweep_entity_id (ezdxf.entities.SweptSurface.dxf attribute), 235
SweptSurface (class in ezdxf.entities), 235

table (class in ezdxf.sections.table), 124
table_entries_to_code() (in module ezdxf.addons.dxf2code), 338
tables (ezdxf.render.Drawing attribute), 112
TablesSection (class in ezdxf.sections.tables), 119
tag (ezdxf.entities.Attdxf.dxf attribute), 151
tag (ezdxf.entities.Attib.dxf attribute), 150
tag_index() (ezdxf.lldxf.tags.Tags method), 428
tagArray (class in ezdxf.lldxf.packedtags), 431
TagList (class in ezdxf.lldxf.packedtags), 431
tags (class in ezdxf.lldxf.dxf), 427
tags (ezdxf.entities.XRecord attribute), 253
tangent (ezdxf.entities.Init.dxf attribute), 225
tangent() (ezdxf.math.Bezier4P method), 294
tangent() (ezdxf.math.ConstructionCircle method), 282
tangent() (ezdxf.math.EulerSpiral method), 295
tangents() (ezdxf.math.ConstructionArc method), 284
target_point (ezdxf.entities.View.dxf attribute), 140
target_point (ezdxf.entities.VPort.dxf attribute), 139
target_vertices (ezdxf.entities.GeoData attribute), 249
temp_path (in module ezdxfaddons.odafc), 362
text (class in ezdxf.entities), 236
text (ezdxf.entities.Attdxf.dxf attribute), 151
text (ezdxf.entities.Attib.dxf attribute), 150
text (ezdxf.entities.Dimension.dxf attribute), 188
text (ezdxf.entities.MText attribute), 217
text (ezdxf.entities.Text.dxf attribute), 236
text_direction (ezdxf.entities.MText.dxf attribute), 216
text_generation_flag (ezdxf.entities.Text.dxf attribute), 237
text_height (ezdxf.entities.Leader.dxf attribute), 208
text_midpoint (ezdxf.entities.Dimension.dxf attribute), 187
text_rotation (ezdxf.entities.Dimension.dxf attribute), 188
text_width (ezdxf.entities.Leader.dxf attribute), 208
Textstyle (class in ezdxf.entities), 129
t.thaw() (ezdxf.entities.Layer method), 128
thickness (ezdxf.entities.DXGGraphic.dxf attribute), 181
thickness (ezdxf.entities.Line.dxf attribute), 209
time (ezdxf.entities.Sun.dxf attribute), 251
tint (ezdxf.entities.Gradient attribute), 205
to2d() (ezdxf.math.Bezier4P method), 293
to3d() (ezdxf.math.Bezier4P method), 294
to_ellipse() (ezdxf.entities.Arc method), 183
to_ellipse() (ezdxf.entities.Circle method), 185
to_ocs() (ezdxf.math.ConstructionEllipse method), 286
ezdxf Documentation, Release 0.13

Index

to_ocs() (ezdxf.math.UCS method), 268
to_ocs_angle_deg() (ezdxf.math.UCS method), 268
to_spline() (ezdxf.entities.Arc method), 183
to_spline() (ezdxf.entities.Circle method), 185
to_spline() (ezdxf.entities.Ellipse method), 195
to_wcs() (ezdxf.math.OCS method), 267
to_wcs() (ezdxf.math.WCS method), 268
tobytes() (ezdxf.entities.Body method), 184
tostring() (ezdxf.entities.Body method), 184
tostring() (ezdxf.lldxf.types.DXFBinaryTag method), 427
Trace (class in ezdxf.entities), 238
transform() (ezdxf.entities.Arc method), 183
transform() (ezdxf.entities.Circle method), 185
transform() (ezdxf.entities.DimText method), 189
transform() (ezdxf.entities.DXFGraphic method), 179
transform() (ezdxf.entities.Ellipse method), 195
transform() (ezdxf.entities.Face3d method), 182
transform() (ezdxf.entities.Hatch method), 199
transform() (ezdxf.entities.Image method), 207
transform() (ezdxf.entities.Insert method), 149
transform() (ezdxf.entities.Leader method), 209
transform() (ezdxf.entities.Line method), 209
transform() (ezdxf.entities.LWPolyline method), 213
transform() (ezdxf.entities.Mesh method), 214
transform() (ezdxf.entities.MeshPolyline method), 218
transform() (ezdxf.entities.MeshRay method), 220
transform() (ezdxf.entities.MeshText method), 224
transform() (ezdxf.entities.Mesh acad method), 227
transform() (ezdxf.entities.Solid method), 228
transform() (ezdxf.entities.SolidLine method), 229
transform() (ezdxf.entities.Spline method), 231
transform() (ezdxf.entities.Text method), 237
transform() (ezdxf.entities.Trace method), 238
transform() (ezdxf.entities.XLine method), 244
transform() (ezdxf.math.BSpline method), 292
transform() (ezdxf.math.ConstructionEllipse method), 287
transform() (ezdxf.math.Matrix44 method), 273
transform() (ezdxf.math.Matrix44 method), 322
transform() (ezdxf.math.Matrix44 method), 273
transform() (ezdxf.math.Matrix44 method), 273
transform() (ezdxf.math.Matrix44 method), 273
transform() (ezdxf.math.Matrix44 method), 273
transformation_matrix_extruded_entity (ezdxf.entities.ExtrudedSurface method), 233
transformation_matrix_path_entity (ezdxf.entities.SweptSurface method), 235
transformation_matrix_revolved_entity (ezdxf.entities.RevolvedSurface method), 234
transformation_matrix_sweep_entity (ezdxf.entities.SweptSurface method), 235
translate() (ezdxf.entities.Circle method), 185
translate() (ezdxf.entities.DXFGraphic method), 179
translate() (ezdxf.entities.Ellipse method), 195
translate() (ezdxf.entities.Insert method), 149
translate() (ezdxf.entities.Line method), 209
translate() (ezdxf.entities.Point method), 220
translate() (ezdxf.entities.Ray method), 227
translate() (ezdxf.entities.Text method), 238
translate() (ezdxf.entities.XLine method), 244
translate() (ezdxf.math.ConstructionArc method), 284
translate() (ezdxf.math.ConstructionBox method), 288
translate() (ezdxf.math.ConstructionCircle method), 282
translate() (ezdxf.math.ConstructionLine method), 281
translate() (ezdxf.math.Matrix44 class method), 271
translate() (ezdxf.math.Shape2d method), 290
translate() (ezdxf.render.MeshTransformer method), 322
transparency (ezdxf.entities.DXFGraphic attribute), 178
transparency (ezdxf.entities.DXFGraphic.dxf attribute), 181
transparency (ezdxf.entities.Layer attribute), 128
transparency (ezdxf.entities.LText attribute), 217
transpose2float() (in module ezdxf.tools), 302
transpose() (ezdxf.math.Matrix method), 299
transpose() (ezdxf.math.Matrix44 method), 273
tridiagonal_matrix_solver() (in module ezdxf.math), 297
tridiagonal_vector_solver() (in module ezdxf.math), 296
true_color, 434
true_color (ezdxf.entities.DXFGraphic.dxf attribute), 181
true_color (ezdxf.entities.Layer.dxf attribute), 127
true_color (ezdxf.entities.LText attribute), 251
tuples_to_tags() (in module ezdxf.lldxf.types), 426
twist_angle (ezdxf.entities.ExtrudedSurface.dxf attribute), 233
twist_angle (ezdxf.entities.RevolvedSurface.dxf attribute), 234
twist_angle (ezdxf.entities.SweptSurface.dxf attribute), 235
u_count (ezdxf.entities.Surface.dxf attribute), 232
u_pixel (ezdxf.entities.Image.dxf attribute), 206
UCS (class in ezdxf.math), 267
ucs (ezdxf.drawing.Drawing attribute), 113
ucss (ezdxf.entities.View.dxf attribute), 141
ucss (ezdxf.sections.tables.TablesSection attribute), 119
) (ezdxf.entities.Insert method), 150
) (ezdxf.entities.UCSTable method), 143
) (ezdxf.math.Matrix4x4 static method), 272
ucs_base_handle (ezdxf.entities.Viewport.dxf attribute), 243
ucs_handle (ezdxf.entities.View.dxf attribute), 141
ucs_handle (ezdxf.entities.Viewport.dxf attribute), 243
ucs_icon (ezdxf.entities.Viewport.dxf attribute), 243
ucs_icon (ezdxf.entities.VPort.dxf attribute), 139
ucs_origin (ezdxf.entities.View.dxf attribute), 141
ucs_origin (ezdxf.entities.Viewport.dxf attribute), 243
ucs_ortho_type (ezdxf.entities.View.dxf attribute), 141
ucs_ortho_type (ezdxf.entities.Viewport.dxf attribute), 243
ucs_x_axis (ezdxf.entities.Viewport.dxf attribute), 243
ucs_xaxis (ezdxf.entities.View.dxf attribute), 141
ucs_y_axis (ezdxf.entities.Viewport.dxf attribute), 243
ucs_yaxis (ezdxf.entities.View.dxf attribute), 141
UCSTable (class in ezdxf.entities), 142
Underlay (class in ezdxf.entities), 239
underlay_def (ezdxf.entities.Underlay attribute), 240
underlay_def (ezdxf.entities.UnderlayDefinition attribute), 239
UnderlayDefinition (class in ezdxf.entities), 252
uniform_knot_vector () (in module ezdxf.math), 257
union () (ezdxf.addons.pycsg.CSG method), 353
unit_vector (ezdxf.entities.Ray.dxf attribute), 227
unit_vector (ezdxf.entities.XLine.dxf attribute), 244
units (ezdxf.entities.BlockRecord.dxf attribute), 143
unlink_entity () (ezdxf.layouts.BaseLayout method), 154
unlink_from_layout ()
(ezdxf.entities.DXFGraphic method), 178
unlock () (ezdxf.entities.Layer method), 128
unnamed (ezdxf.entities.dxffgroups.DXFGroup.dxf attribute), 173
up_direction (ezdxf.entities.GeoData.dxf attribute), 248
update () (ezdxf.entities.DimStyleOverride method), 190
update () (ezdxf.llidxf.tags.Tags method), 428
update_dxf_attribs () (ezdxf.entities.DXFEntity method), 176
update_instance_counters ()
(ezdxf.sections.classes.ClassesSection method), 118
update_paper () (ezdxf.layouts.ClassesSection method), 170
upper (ezdxf.math.BandedMatrixLU attribute), 300
upper_right (ezdxf.entities.VPort.dxf attribute), 139
use_plot_styles () (ezdxf.layouts.Layout method), 170
use_standard_scale () (ezdxf.layouts.Layout method), 170
user_location_override ()
(ezdxf.entities.DimStyleOverride method), 192
user_scale_factor (ezdxf.entities.GeoData.dxf attribute), 248
ux (ezdxf.math.OCs attribute), 267
ux (ezdxf.math.UCS attribute), 268
uy (ezdxf.math.OCs attribute), 267
uy (ezdxf.math.UCS attribute), 268
uz (ezdxf.math.OCs attribute), 267
uz (ezdxf.math.UCS attribute), 268
V
v_count (ezdxf.entities.Surface.dxf attribute), 232
v_pixel (ezdxf.entities.Image.dxf attribute), 206
validate () (ezdxf.drawing.Drawing method), 116
valign (ezdxf.entities.Text.dxf attribute), 236
value (ezdxf.entities.dxf attribute), 247
values (ezdxf.lldxf.packedtags.TagArray attribute), 431
values (ezdxf.lldxf.packedtags.TagList attribute), 431
values () (ezdxf.entitydb.EntityDB method), 424
varnames () (ezdxf.sections.header.HeaderSection method), 117
Vec2 (class in ezdxf.math), 278
vec2 (ezdxf.math.Vector attribute), 274
Vector (class in ezdxf.math), 274
vector (ezdxf.math.Plane attribute), 278
version (ezdxf.entities.Body.dxf attribute), 184
version (ezdxf.entities.Dimension.dxf attribute), 186
version (ezdxf.entities.GeoData.dxf attribute), 247
version (ezdxf.entities.Mesh.dxf attribute), 214
version (ezdxf.entities.Sun.dxf attribute), 251
Vertex (class in ezdxf.entities), 224
VERTEX_SIZE (ezdxf.lldxf.packedtags.VertexArray attribute), 431
VertexArray (class in ezdxf.lldxf.packedtags), 431
vertical_unit_scale (ezdxf.entities.GeoData.dxf attribute), 248
vertical_units (ezdxf.entities.GeoData.dxf attribute), 248
vertices (ezdxf.entities.Leader attribute), 209
vertices (ezdxf.entities.Mesh attribute), 214
vertices (ezdxf.entities.MeshData attribute), 214
vertices (ezdxf.entities.MeshVertexCache attribute), 226
vertices (ezdxf.entities.Polyline attribute), 222
vertices (ezdxf.entities.PolylinePath attribute), 201
vertices (ezdxf.math.Shape2d attribute), 289
vertices (ezdxf.render.MeshBuilder attribute), 320
vertices() (ezdxf.entities.Circle method), 185
vertices() (ezdxf.entities.Ellipse method), 195
vertices() (ezdxf.entities.LWPolyline method), 212
vertices() (ezdxf.math.ConstructionArc method), 284
vertices() (ezdxf.math.ConstructionEllipse method), 286
vertices_in_wcs() (ezdxf.entities.LWPolyline method), 212
View (class in ezdxf.entities), 140
view_brightness (ezdxf.entities.Viewport.dxf attribute), 244
view_center_point (ezdxf.entities.Viewport.dxf attribute), 241
view_contrast (ezdxf.entities.Viewport.dxf attribute), 244
view_direction_vector (ezdxf.entities.Viewport.dxf attribute), 241
view_height (ezdxf.entities.Viewport.dxf attribute), 242
view_mode (ezdxf.entities.View.dxf attribute), 140
view_mode (ezdxf.entities.VPort.dxf attribute), 139
view_target_point (ezdxf.entities.Viewport.dxf attribute), 241
view_twist (ezdxf.entities.View.dxf attribute), 140
view_twist (ezdxf.entities.VPort.dxf attribute), 139
view_twist_angle (ezdxf.entities.Viewport.dxf attribute), 242
Viewport (class in ezdxf.entities), 241
viewporters (ezdxf.drawing.Drawing attribute), 113
viewporters (ezdxf.sections.tables.TablesSection attribute), 120
viewporters() (ezdxf.layouts.Paperspace method), 171
ViewportTable (class in ezdxf.sections.table), 126
views (ezdxf.drawing.Drawing attribute), 113
views (ezdxf.sections.tables.TablesSection attribute), 120
virtual_entities() (ezdxf.entities.Dimension method), 189
virtual_entities() (ezdxf.entities.Insert method), 149
virtual_entities() (ezdxf.entities.LWPolyline method), 213
virtual_entities() (ezdxf.entities.Polyline method), 224
virtual_guide (ezdxf.entities.LoftedSurface.dxf attribute), 234
virtual_pen_number (ezdxf.addons.acadctb.PlotStyle attribute), 342
visual_style_handle (ezdxf.entities.View.dxf attribute), 141
visual_style_handle (ezdxf.entities.Viewport.dxf attribute), 243
VPort (class in ezdxf.entities), 138
vtx0 (ezdxf.entities.Face3d.dxf attribute), 181
vtx0 (ezdxf.entities.Solid.dxf attribute), 229
vtx0 (ezdxf.entities.Trace.dxf attribute), 238
vtx1 (ezdxf.entities.Face3d.dxf attribute), 181
vtx1 (ezdxf.entities.Solid.dxf attribute), 229
vtx1 (ezdxf.entities.Trace.dxf attribute), 238
vtx1 (ezdxf.entities.Vertex.dxf attribute), 225
vtx2 (ezdxf.entities.Face3d.dxf attribute), 182
vtx2 (ezdxf.entities.Solid.dxf attribute), 229
vtx2 (ezdxf.entities.Trace.dxf attribute), 238
vtx2 (ezdxf.entities.Vertex.dxf attribute), 225
vtx3 (ezdxf.entities.Face3d.dxf attribute), 182
vtx3 (ezdxf.entities.Solid.dxf attribute), 229
vtx3 (ezdxf.entities.Trace.dxf attribute), 238
vtx3 (ezdxf.entities.Vertex.dxf attribute), 225
vtx4 (ezdxf.entities.Vertex.dxf attribute), 225

W
was_a_proxy (ezdxf.entities.DXFClass.dxf attribute), 119
weights (ezdxf.entities.Spline attribute), 231
weights (ezdxf.entities.SplineEdge attribute), 204
weights() (ezdxf.math.BSpline method), 291
width (ezdxf.entities.MText.dxf attribute), 215
width (ezdxf.entities.Text.dxf attribute), 236
width (ezdxf.entities.Textstyle.dxf attribute), 129
width (ezdxf.entities.View.dxf attribute), 140
width (ezdxf.entities.Viewport.dxf attribute), 241
width (ezdxf.math.ConstructionBox attribute), 288
write() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 341
write() (ezdxf.addons.acadctb.NamedPlotStyles method), 342
write() (ezdxf.addons.iterdxf.IterDXFWriter method), 332
write() (ezdxf.drawing.Drawing method), 114
write_fixed_meta_data_for_testing (in module ezdxf.options), 301

X
x (ezdxf.math.Vector attribute), 274
X_AXIS (in module ezdxf.math), 278
x_rotate() (ezdxf.math.Matrix44 class method), 271
xaxis (ezdxf.entities.UCSTable.dxf attribute), 143
xdata (ezdxf.lldxf.extendedtags.ExtendedTags attribute), 429
XLine (class in ezdxf.entities), 244
xof() (ezdxf.math.ConstructionRay method), 281
XRecord (class in ezdxf.entities), 253
xref_path (ezdxf.entities.Block.dxf attribute), 145
xround() (in module ezdxf.math), 257
xscale (ezdxf.entities.Insert.dxf attribute), 146
xscale (ezdxf.entities.Shape.dxf attribute), 228
xy (ezdxf.math.Vector attribute), 274
xyz (ezdxf.math.Vector attribute), 274
xyz_rotate() (ezdxf.math.Matrix44 class method), 272

Y
y (ezdxf.math.Vector attribute), 274
Y_AXIS (in module ezdxf.math), 278
y_rotate() (ezdxf.math.Matrix44 class method), 271
yaxis (ezdxf.entities.UCSTable.dxf attribute), 143
yof() (ezdxf.math.ConstructionRay method), 281
yscale (ezdxf.entities.Insert.dxf attribute), 146

Z
z (ezdxf.math.Vector attribute), 274
Z_AXIS (in module ezdxf.math), 278
z_rotate() (ezdxf.math.Matrix44 class method), 271
zoom_to_paper_on_update() (ezdxf.layouts.Layout method), 170
zorder (ezdxf.entities.DXFGraphic attribute), 178
zscale (ezdxf.entities.Insert.dxf attribute), 146