
ezdxf Documentation

Release 0.8.9

Manfred Moitzi

Oct 27, 2018

Contents

1	Website	3
2	Documentation	5
3	Questions and Feedback at Google Groups	7
4	Contents	9
4.1	Introduction	9
4.2	Tutorials	10
4.3	Reference	37
4.4	Add-ons	142
4.5	Howto	151
4.6	DXF Internals	153
5	News	201
5.1	Indices and tables	206
	Python Module Index	207

Welcome! This is the documentation for ezdxf 0.8.9, last updated Oct 27, 2018.

Note: **Python 2** support will be dropped in *ezdxf* v0.9.0, because **Python 2** support get more and more annoying. *ezdxf* v0.9.0 requires at least **Python 3.6**, because I want f-strings!

- *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 CPython 3.6, because of more and more troubles with Python 2 support for Python 2 will be dropped in *ezdxf* v0.9.0
- OS independent
- additional required packages: [pyparsing](#)
- MIT-License
- read/write/new support for DXF versions: R12, R2000, R2004, R2007, R2010, R2013 and R2018
- additional read support for DXF versions R13/R14 (upgraded to R2000)
- additional read support for older DXF versions than R12 (upgraded to R12)
- preserves third-party DXF content
- additional *Fast DXF R12 File/Stream Writer*, that creates just an ENTITIES section with support for the basic DXF entities

CHAPTER 1

Website

<https://ezdxf.mozman.at/>

CHAPTER 2

Documentation

Documentation of development version at <https://ezdx.f.mozman.at/docs>

Documentation of latest release at <http://ezdx.f.readthedocs.io/>

Source Code: <http://github.com/mozman/ezdx.f.git>

Issue Tracker at GitHub: <http://github.com/mozman/ezdx.f/issues>

CHAPTER 3

Questions and Feedback at Google Groups

<https://groups.google.com/d/forum/python-ezdx>

4.1 Introduction

4.1.1 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 all the possibilities of the [DXF](#) format. Nevertheless, a basic understanding of the DXF format is an advantage (but not necessary), also to understand what is possible with the DXF file format and what is not.

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 [dxfgripper](#) packages but with different APIs, both packages are in maintenance only mode, no new features will be added, but they stay available, getting bug fixes and will be adapted for new Python versions.

4.1.2 What ezdxf is NOT

- *ezdxf* is not a DXF converter: *ezdxf* can not convert between different DXF versions, if you are looking for an appropriate program, use *DWG TrueView* from [Autodesk](#), but the latest version can only convert to the DWG format, for converting between DXF versions you need at least AutoCAD LT.
- *ezdxf* is not a CAD file format converter: *ezdxf* can not convert DXF files to **ANY** other format, like SVG, PDF or DWG
- *ezdxf* is not a DXF renderer, it does not create a visual representation of the DXF file content (see above).
- *ezdxf* is not a CAD kernel, *ezdxf* does not provide any 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](#).

4.1.3 Supported Python Versions

ezdxf requires at least Python 3.6. Python 2 support will be dropped in *ezdxf* v0.9.0, because Python 2 support get more and more annoying. I run unit tests with the latest stable CPython 3 version and the latest stable release of *pypy* during development.

ezdxf is written in pure Python and requires only *parser* as additional library beside the Python Standard Library. *pytest* is required to run the provided 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.

4.1.4 Supported Operating Systems

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

4.1.5 Supported DXF Versions

Version	AutoCAD Release
AC1009	AutoCAD R12
AC1012	AutoCAD R13 -> R2000
AC1014	AutoCAD R14 -> R2000
AC1015	AutoCAD R2000
AC1018	AutoCAD R2004
AC1021	AutoCAD R2007
AC1024	AutoCAD R2010
AC1027	AutoCAD R2013
AC1032	AutoCAD R2018

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

4.1.6 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.

4.1.7 License

ezdxf is licensed under the very liberal [MIT-License](#).

4.2 Tutorials

4.2.1 Tutorial for Getting Data from DXF Files

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

At first load the drawing:

```
import ezdxf

dwg = 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 any DXF entity like LINE, CIRCLE, TEXT and so on. Every DXF entity can only reside in exact one layout.

There are three different layout types:

- model space: this is the common construction space
- paper space: used to create print layouts
- block: reusable elements, every block has its own entity space

A DXF drawing consist of exact one model space and at least of one paper space. The DXF12 standard has only one unnamed paper space the later DXF versions support more than one paper space and each paper space has a name.

Iterate over DXF Entities of a Layout

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

```
# iterate over all entities in model space
msp = dwg.modelspace()
for e in msp:
    if e.dxf.type() == 'LINE':
        print_entity(e)

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

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

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

See common DXF attributes:

- *Common DXF Attributes for DXF R12*

- *Common DXF Attributes for DXF R13 or later*

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

```
p = e.get_dxf_attrib('paperspace', 0) # if 'paperspace' is left off, the entity_
↳ defaults to model space
```

An unsupported DXF attribute raises an `DXFAttributeError`.

Getting a Paper Space

```
paperspace = dwg.layout('layout0')
```

Retrieves the paper space named `layout0`, the usage of the layout object is the same as of the model space object. The DXF12 standard provides only one paper space, therefore the paper space name in the method call `dwg.layout('layout0')` is ignored or can be left off. For the later standards you get a list of the names of the available layouts by `Drawing.layout_names()`.

Retrieve Entities by Query Language

Inspired by the wonderful `jQuery` framework, I created a flexible query language for DXF entities. To start a query use the `Layout.query()` method, provided by all sort of layouts 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 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 model space:

```
modelspace = dwg.modelspace()
lines = modelspace.query('LINE')
```

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

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

The `*` is a wildcard for all DXF entities, 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:

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

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

```
all_lines_and_circles_at_the_construction_layer = dwg.entities.query('LINE_
↳ CIRCLE[layer=="construction"]')
```

Get all model space entities at layer `construction`, but no entities with the `linestyle` `DASHED`:


```
not_dashed_entities = modelspace.query('*[layer=="construction" and linestyle!="DASHED
↪"]')
```

Retrieve Entities by groupby

TODO

Default Layer Settings

See also:

Tutorial for Layers and class *Layer*

4.2.2 Tutorial for Creating Simple DXF Drawings

Fast DXF R12 File/Stream Writer - 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:

Fast DXF R12 File/Stream Writer

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

```
import ezdxf

dwg = ezdxf.new('R2010') # create a new DXF R2010 drawing, official DXF version_
↪name: 'AC1024'

msp = dwg.modelspace() # add new entities to the model space
msp.add_line((0, 0), (10, 0)) # add a LINE entity
dwg.saveas('line.dxf')
```

New entities are always added to layouts, a layout can be the model space, a paper space layout or a block layout.

See also:

Look at the *Layout* factory methods to see all the available DXF entities.

4.2.3 Tutorial for Layers

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

First you have to create layers, assigning them names and properties such as color and linetype. Then you can assign those layers to other drawing entities. To assign a layer just use its name as string. It is not recommend but it is possible

to use layers without a layer definition, just use the layer name without a definition, the layer has the default linetype *Continuous* and the default color is *1*.

Create a new layer definition:

```
import ezdxf

dwg = ezdxf.new()
msp = modelspace()
dwg.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:

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

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

Changing Layer State

First get the layer definition object:

```
my_lines = dwg.layers.get('MyLines')
```

Now you check the state of the layer:

```
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
```

And you can change the state of the layer:

```
my_lines.off() # switch layer off, will not shown in CAD programs/viewers
my_lines.lock() # layer is not editable in CAD programs
```

Setting/Getting the default color of the layer should be done with `Layer.get_color()` and `Layer.set_color()` because the color value is misused for switching the layer on and off, layer is off if the color value is negative.

Changing the default layer values:

```
my_lines.dxf.linetype = 'DOTTED'
my_lines.set_color(13) # preserves the layer on/off state
```

See also:

for all methods and attributes see class `Layer`.

Check Available Layers

The layers object supports some standard Python protocols:

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

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

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

Deleting a Layer

You can delete a layer definition:

```
dwg.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*.

4.2.4 Tutorial for Blocks

What are Blocks?

Blocks are reusable elements, you can see it as container for other DXF entities which can be placed multiply times on different places. But instead of inserting the DXF entities of the block several times just a block reference is placed.

Create a Block

Blocks are managed by the *BlocksSection* class and every drawing has only one blocks section: *Drawing.blocks*.

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

def get_random_point():
    """Creates 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
dwg = ezdxf.new('ac1024')

# Create a block with the name 'FLAG'
flag = dwg.blocks.new(name='FLAG')

# Add DXF entities to the block 'FLAG'.
# The default base point (= insertion point) of the block is (0, 0).
flag.add_polyline2d([(0, 0), (0, 5), (4, 3), (0, 3)]) # the flag as 2D polyline
flag.add_circle((0, 0), .4, dxfattribs={'color': 2}) # mark the base point with a_
↪circle
```

(continues on next page)

Insert a Block

A block reference is a DXF *Insert* entity and can be placed in any *Layout: Model Space*, any *Paper Space* or a *BlockLayout* (which enables blocks in blocks). Every block reference can be scaled and rotated individually.

Lets insert some random flags into the modelspace:

```
# Get the modelspace of the drawing.
modelspace = dwg.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'.
    modelspace.add_blockref('FLAG', point, dxfattribs={
        'xscale': random_scale,
        'yscale': random_scale,
        'rotation': -15
    })

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

What are Attributes?

An attribute (*Attrib*) is a text annotation to 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, 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_auto_blockref()* ('auto' is for automatically filled attributes). 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 it has the disadvantage, that the block reference is wrapped into an anonymous block, which makes evaluation of attributes more complex.

Using attribute definitions (*Attdef*):

```
# 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), {'height': 0.5, 'color': 3})
flag.add_attdef('XPOS', (0.5, -1.0), {'height': 0.25, 'color': 4})
flag.add_attdef('YPOS', (0.5, -1.5), {'height': 0.25, 'color': 4})
```

(continues on next page)

(continued from previous page)

```

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

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

    # Every flag has a different scaling and a rotation of +15 deg.
    random_scale = 0.5 + random.random() * 2.0
    modelspace.add_auto_blockref('FLAG', point, values, dxfattribs={
        'xscale': random_scale,
        'yscale': random_scale,
        'rotation': 15
    })

# Save the drawing.
dwg.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:

```

# 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 = dwg.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)

```

4.2.5 Tutorial for LWPolyline

A lightweight polyline is defined as a single graphic entity. The *LWPolyline* differs from the old-style *Polyline*, which is defined as a group of subentities. *LWPolyline* display faster (in AutoCAD) and consume less disk space and RAM. LWPolylines are planar elements, therefore all coordinates have no value for the z axis.

Create a simple polyline:

```
import ezdxf

dwg = ezdxf.new('AC1015')
msp = dwg.modelspace()

points = [(0, 0), (3, 0), (6, 3), (6, 6)]
msp.add_lwpolyline(points)

dwg.saveas("lwpolyline1.dxf")
```

Append points to a polyline:

```
dwg = ezdxf.readfile("lwpolyline1.dxf")
msp = dwg.modelspace()

line = msp.query('LWPOLYLINE')[0] # take first LWPolyline
line.append_points([(8, 7), (10, 7)])

dwg.saveas("lwpolyline2.dxf")
```

Getting points always returns a 5-tuple (x, y, start_width, end_width, bulge), start_width, end_width and bulge is 0 if not present (0 is the DXF default value if not present):

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

Use context manager to edit polyline:

```
dwg = ezdxf.readfile("lwpolyline2.dxf")
msp = dwg.modelspace()

line = msp.query('LWPOLYLINE')[0] # take first LWPolyline

with line.points() 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)]
# implicit call of line.set_points(points) at context manager exit

dwg.saveas("lwpolyline3.dxf")
```

Each line segment can have a different start/end width, if omitted start/end width = 0:

```
dwg = ezdxf.new('AC1015')
msp = dwg.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)]
```

(continues on next page)

(continued from previous page)

```
msp.add_lwpolyline(points)

dwg.saveas("lwpolyline4.dxf")
```

The first vertex (point) carries the start/end width of the first segment, the second vertex of the second segment and so on, the start/end width value of the last vertex is ignored. Start/end width only works if the DXF attribute *const_width* is unset, to be sure delete it:

```
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:

```
dwg = ezdxf.new('AC1015')
msp = dwg.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)

dwg.saveas("lwpolyline5.dxf")
```

The curved segment is drawn from the vertex with the defined *bulge* value to the following vertex, the curved segment is always a circle, the diameter is relative to the vertex distance, *bulge* = 1.0 means the diameter equals the vertex distance, *bulge* = 0.5 means the diameter is the half of the vertex distance. *bulge* > 0 the curve is on the right side of the vertex connection line, *bulge* < 0 the curve is on the left side.



4.2.6 Tutorial for Text

TEXT - just one line

Add simple one line text with the factory function `Layout.add_text()`.

```
import ezdxf

dwg = ezdxf.new('AC1009') # TEXT is a basic entity and exists in every DXF standard

msp = dwg.modelspace()

# use set_pos() for proper TEXT alignment - the relations between halign, valign, ↵
↵insert and align_point are tricky.
```

(continues on next page)

(continued from previous page)

```
msp.add_text("A Simple Text").set_pos((2, 3), align='MIDDLE_RIGHT')

# using text styles
dwg.styles.new('custom', dxfattribs={'font': 'times.ttf', 'width': 0.8}) # Arial, ↵
↵default width factor of 0.8
msp.add_text("Text Style Example: Times New Roman", dxfattribs={'style': 'custom',
↵'height': 0.35}).set_pos((2, 6), align='LEFT')

dwg.saveas("simple_text.dxf")
```

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

Vert/Horiz	Left	Center	Right
Top	TOP_LEFT	TOP_CENTER	TOP_RIGHT
Middle	MIDDLE_LEFT	MIDDLE_CENTER	MIDDLE_RIGHT
Bottom	BOTTOM_LEFT	BOTTOM_CENTER	BOTTOM_RIGHT
Baseline	LEFT	CENTER	RIGHT

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.

Align-ment	Description
ALIGNED	Text is stretched or compressed to fit exactly between <i>p1</i> and <i>p2</i> and the text height is also adjusted to preserve height/width ratio.
FIT	Text is stretched or compressed to fit exactly between <i>p1</i> and <i>p2</i> but only the text width is adjusted, the text height is fixed by the <i>height</i> attribute.
MIDDLE	also a <i>special</i> adjustment, but the result is the same as for <i>MIDDLE_CENTER</i> .

more is coming soon ...

4.2.7 Tutorial for MText

coming soon ...

4.2.8 Tutorial for Spline

Create a simple spline:

```
import ezdxf

dwg = ezdxf.new('AC1015') # splines requires the DXF R2000 format or later

fit_points = [(0, 0, 0), (750, 500, 0), (1750, 500, 0), (2250, 1250, 0)]
msp = dwg.modelspace()
msp.add_spline(fit_points)

dwg.saveas("simple_spline.dxf")
```

Add a fit point to a spline:


```

import ezdxf

dwg = ezdxf.readfile("simple_spline.dxf")

msp = dwg.modelspace()
spline = msp.query('SPLINE')[0] # take the first spline

# use the context manager
with spline.edit_data() as data: # data contains standard python lists
    data.fit_points.append((2250, 2500, 0))

    points = data.fit_points[:-1] # pitfall: this creates a new list without a
    ↪connection to the spline object
    points.append((3000, 3000, 0)) # has no effect for the spline object

    data.fit_points = points # replace points of fp, this way it works

# the context manager calls automatically spline.set_fit_points(data.fit_points)

dwg.saveas("extended_spline.dxf")

```

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 an AutoCAD generated file:

```

import ezdxf

dwg = ezdxf.readfile("AutoCAD_generated.dxf")

msp = dwg.modelspace()
spline = msp.query('SPLINE')[0] # take the first spline
with spline.edit_data() as data: # context manager
    data.fit_points.append((2250, 2500, 0)) # data.fit_points is a standard python
    ↪list

    # As far as I tested this works without complaints from AutoCAD, but for the case
    ↪of problems
    data.knot_values = [] # delete knot values, this could modify the geometry of
    ↪the spline
    data.weights = [] # delete weights, this could modify the geometry of the spline
    data.control_points = [] # delete control points, this could modify the geometry
    ↪of the spline

dwg.saveas("modified_spline.dxf")

```

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

```

if spline.closed:
    # this spline is closed
    pass

# close a spline
spline.closed = True

# open a spline
spline.closed = False

```

Set start/end tangent:

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

Get count of fit points:

```
# as stored in the DXF file
count = spline.dxf.n_fit_points
# or count by yourself
count = len(spline.get_fit_points())
```

4.2.9 Tutorial for Polyface

coming soon ...

4.2.10 Tutorial for Mesh

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

```
import ezdxf

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

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

dwg = ezdxf.new('AC1015') # mesh requires the DXF 2000 or newer format
msp = dwg.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

dwg.saveas("cube_mesh_1.dxf")
```

Create a cube mesh by method calls:

```

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),
]

dwg = ezdxf.new('AC1015') # mesh requires the DXF 2000 or newer format
msp = dwg.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

dwg.saveas("cube_mesh_2.dxf")

```

4.2.11 Tutorial for Hatch

Create hatches with one boundary path

The simplest form of a hatch has one polyline path with only straight lines as boundary path:

```

import ezdxf

dwg = ezdxf.new('AC1015') # hatch requires the DXF R2000 (AC1015) format or later
msp = dwg.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)
with hatch.edit_boundary() as boundary: # edit boundary path (context manager)
    # 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
    boundary.add_polyline_path([(0, 0), (10, 0), (10, 10), (0, 10)], is_closed=1)

dwg.saveas("solid_hatch_polyline_path.dxf")

```

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

```
import ezdxf
```

(continues on next page)

(continued from previous page)

```

dwg = ezdxf.new('AC1015') # hatch requires the DXF R2000 (AC1015) format or later
msp = dwg.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)
with hatch.edit_boundary() as boundary: # edit boundary path (context manager)
    # 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
    boundary.add_polyline_path([(0, 0, 1), (10, 0), (10, 10, -0.5), (0, 10)], is_
↳closed=1)

dwg.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:

```

import ezdxf

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

# important: major axis >= minor axis (ratio <= 1.)
msp.add_ellipse((0, 0), major_axis=(0, 10), ratio=0.5) # minor axis length = major_
↳axis length * ratio

hatch = msp.add_hatch(color=2) # by default a solid fill hatch with fill color=7
↳(white/black)
with hatch.edit_boundary() as boundary: # edit boundary path (context manager)
    # every boundary path is always a 2D element
    edge_path = boundary.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)

dwg.saveas("solid_hatch_ellipse.dxf")

```

Create hatches with multiple boundary paths (islands)

TODO

Create hatches with with pattern fill

TODO

Create hatches with gradient fill

TODO

4.2.12 Tutorial for Hatch Pattern Definition

TODO

4.2.13 Tutorial for Image and ImageDef

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

```
import ezdxf

dwg = ezdxf.new('AC1015') # image requires the DXF R2000 format or later
my_image_def = dwg.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 = dwg.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 = dwg.objects.query('IMAGEDEF') # get all image defs in drawing

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

4.2.14 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:

```
import ezdxf

dwg = ezdxf.new('AC1015') # underlay requires the DXF R2000 format or later
my_underlay_def = dwg.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 = dwg.modelspace()
# add first underlay
msp.add_underlay(my_underlay_def, insert=(2, 1, 0), scale=0.05)
# The (PDF)UNDERLAY entity is like the INSERT entity, it creates an underlay ↵
↵reference,
```

(continues on next page)

(continued from previous page)

```
# and there can be multiple references to the same underlay in a drawing.
msp.add_underlay(my_underlay_def, insert=(4, 5, 0), scale=.5, rotation=30)

# get existing underlay definitions, Important: UNDERLAYDEFs resides in the objects_
↳section
pdf_defs = dwg.objects.query('PDFDEFINITION') # get all pdf underlay defs in drawing

dwg.saveas("dxf_with_underlay.dxf")
```

4.2.15 Tutorial for Linetypes

Simple line type example:

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

```
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:

```
import ezdxf
from ezdxf.tools.standards import linetypes # some predefined line types

dwg = 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 dwg.linetypes:
        dwg.linetypes.new(name=name, dxfattribs={'description': desc, 'pattern':_
↳pattern})
```

Setup some predefined linetypes:

```
for name, desc, pattern in linetypes():
    if name not in dwg.linetypes:
        dwg.linetypes.new(name=name, dxfattribs={'description': desc, 'pattern':_
↳pattern})
```

Check Available Linetypes

The linetypes object supports some standard Python protocols:

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

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

count = len(dwg.linetypes) # total count of linetypes
```

Removing Linetypes

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

You can delete a linetype:

```
dwg.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.

4.2.16 Tutorial for Complex Linetypes

With 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 — 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:

```
dwg = ezdxf.new('R2018') # DXF R13 or later is required

dwg.linetypes.new('GASLEITUNG2', dxfattribs={
    'description': 'Gasleitung2 ----GAS----GAS----GAS----GAS----GAS----GAS---',
    'length': 1, # required for complex line types
    # line type definition in acadt.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 ... scaling factor, always > 0, if S=0 the TEXT or SHAPE is not visible
- R or U ... rotation relative to the line direction
- X ... x direction offset (along the line)
- Y ... 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:

```
dwg.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.

4.2.17 Tutorial for OCS/UCS Usage

First you need an understanding of vectors, if you don't have it, 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 version 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* has an elevation attribute, *Hatch* has an elevation point (z=elevation, x=y=0), and so on.

Thickness

Defines the extrusion distance for an entity.

Placing 2D Circle in 3D Space

The colors for axis follow the AutoCAD standard:

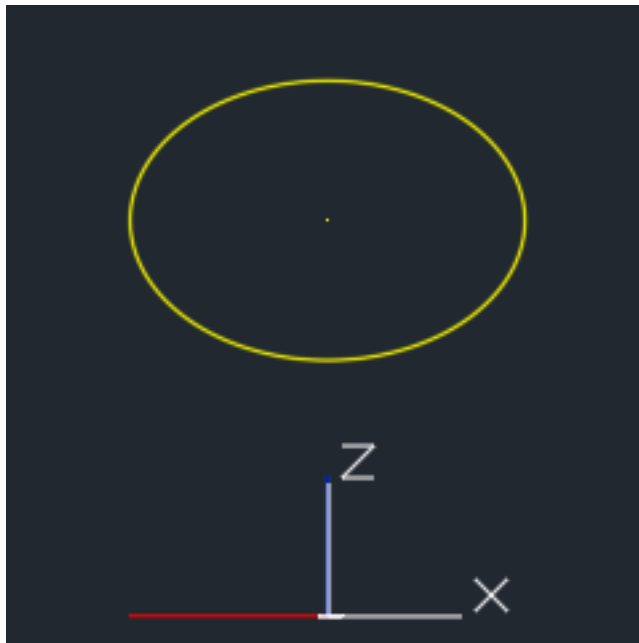
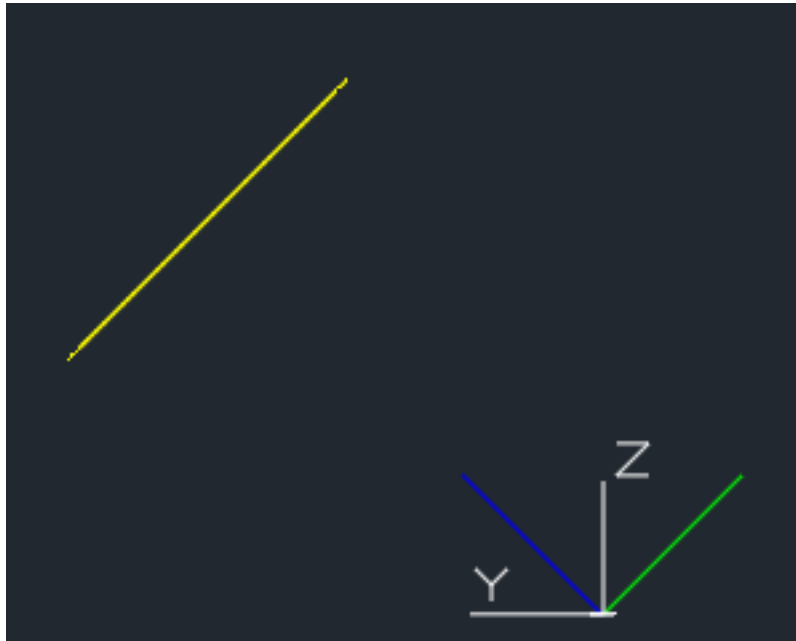
- red is x-axis
- green is y-axis
- blue is z-axis

```
import ezdxf
from ezdxf.algebra import OCS

dwg = ezdxf.new('R2010')
msp = dwg.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={
        # here the extrusion vector should be normalized,
        # which is granted by using the ocs.uz
        'extrusion': ocs.uz,
        'color': 2,
    }
)
# mark center point of circle in WCS
msp.add_point((0, 2, 2), dxfattribs={'color': 2})
```

The following image shows the 2D circle in 3D space in AutoCAD *Left* and *Front* view. The blue line shows the OCS z-axis (extrusion direction), elevation is the distance from the origin to the center of the circle in this case 2.828, and you see that the x- and y- axis of OCS and WCS are not aligned.



Placing LWPolyline in 3D Space

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

```
import ezdxf
from ezdxf.algebra import Vector, UCS
```

(continues on next page)

(continued from previous page)

```

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

# center point 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))

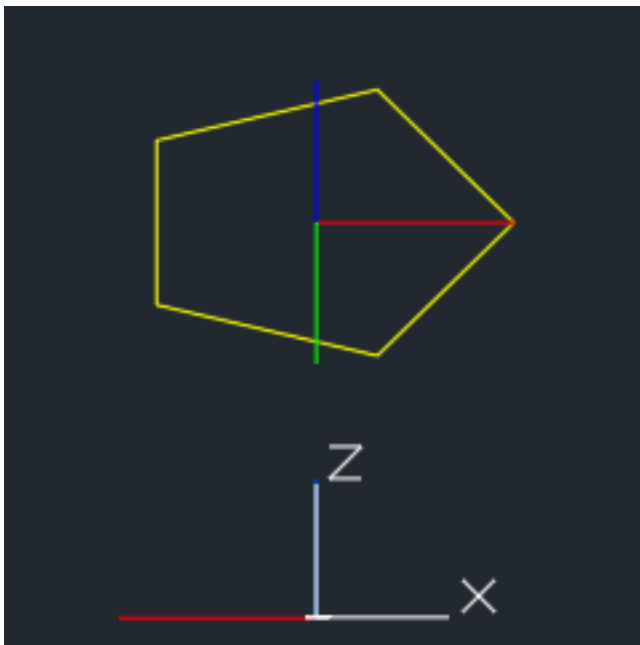
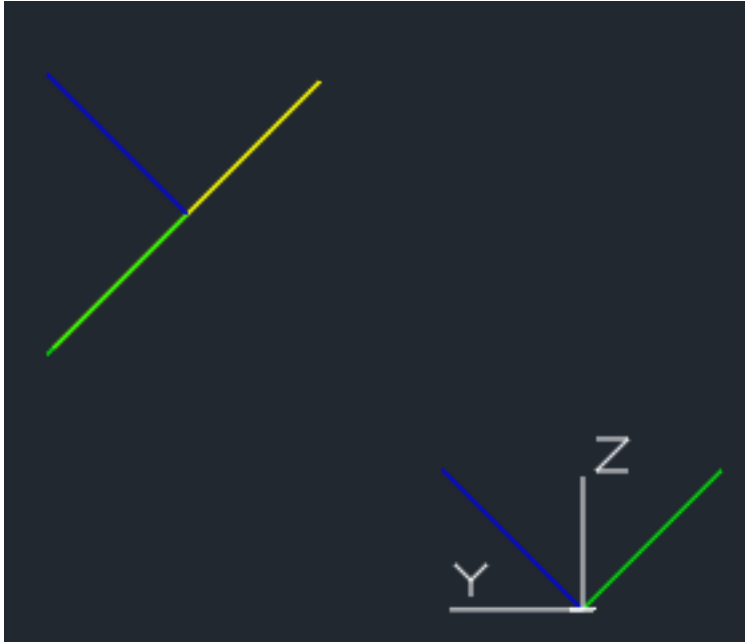
# LWPOLYLINE accepts only 2D points and has an separated DXF attribute elevation.
# All points have the same z-axis (elevation) in OCS!
elevation = ocs_points[0].z

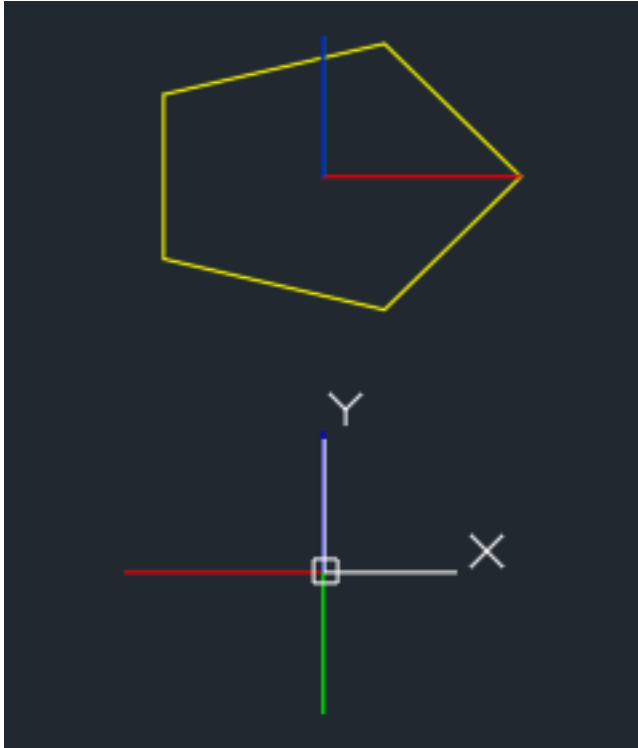
msp.add_lwpolyline(
    # LWPOLYLINE point format: (x, y, [start_width, [end_width, [bulge]]])
    # the z-axis would be start_width, so remove it
    points=[p[:2] for p in ocs_points],
    dxfattribs={
        'elevation': elevation,
        'extrusion': ucs.uz,
        'closed': True,
        'color': 2,
    })

```

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

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





Using UCS to Place 3D Polyline

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

```
import math
import ezdxf
from ezdxf.algebra import UCS, Matrix44

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

# 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
)
transformed_corners_ucs = tmatrix.transform_vectors(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
)
```

(continues on next page)

(continued from previous page)

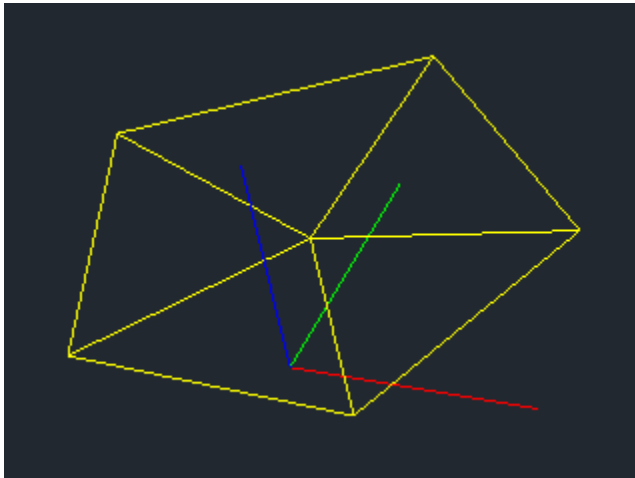
```

corners_wcs = list(ucs.points_to_wcs(transformed_corners_ucs))

msp.add_polyline3d(
    points=corners_wcs,
    dxfattribs={
        'closed': True,
        'color': 2,
    })

# add lines from center to corners
center_wcs = ucs.to_wcs((0, .333, .333))
for corner in corners_wcs:
    msp.add_line(center_wcs, corner, dxfattribs={'color': 2})

```



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 the positive OCS x-axis, and the x-axis is calculated by the *Arbitrary Axis Algorithm*.

Calculate the OCS rotation angle by converting the rotation angle in UCS or WCS into a vector or start 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).

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

```

import ezdxf
from ezdxf.algebra import UCS, Vector

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

# thickness for text works only with shx fonts not with true type fonts
dwg.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

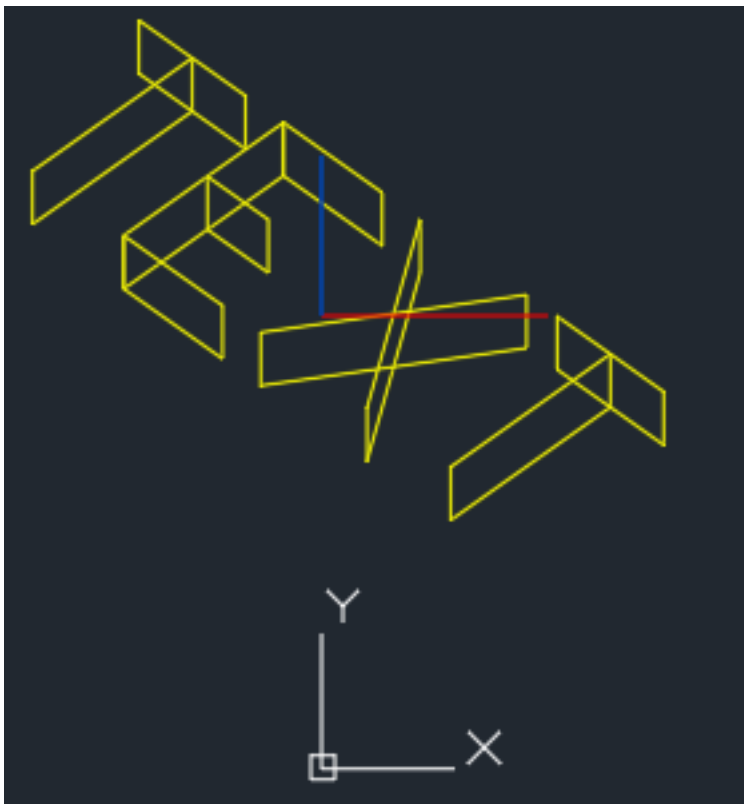
```

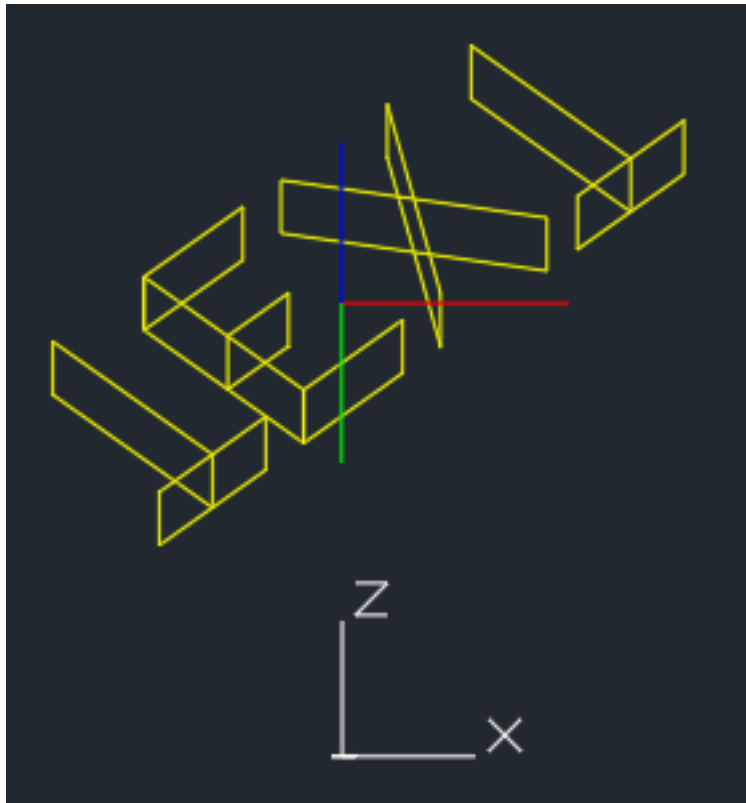
(continues on next page)

(continued from previous page)

```
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': 2,
        '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

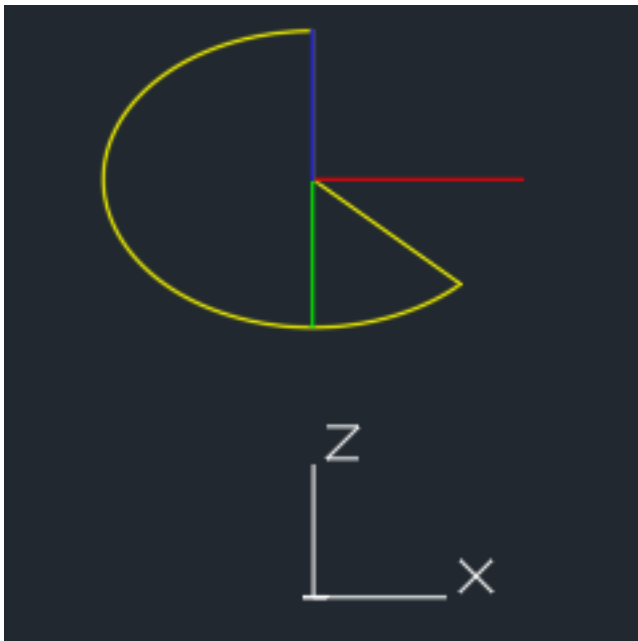
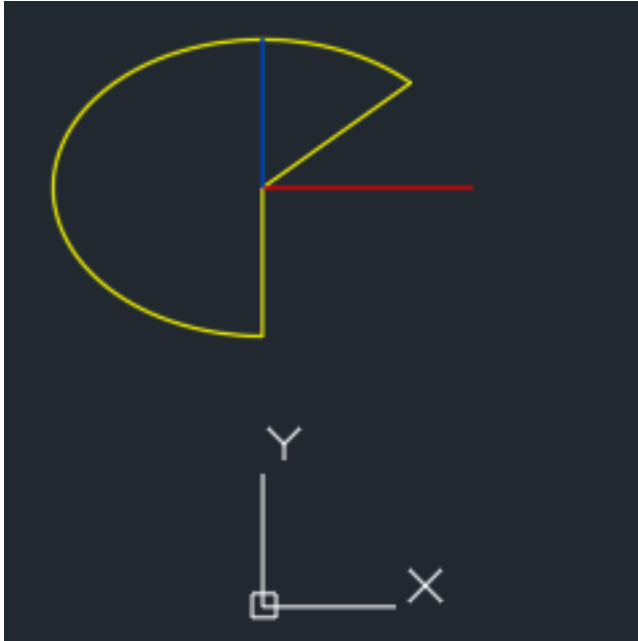
Here we have the same problem as for placing text, you need the start and end angle of the arc in degrees in OCS, and this example also shows a shortcut for calculating the OCS angles.

```
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), # shortcut
    end_angle=ucs.to_ocs_angle_deg(270), # shortcut
    dxfattribs={
        'extrusion': ucs.uz,
        'color': 2,
    })
center = ucs.to_wcs((0, 0))
msp.add_line(
    start=center,
    end=ucs.to_wcs(Vector.from_deg_angle(45)),
    dxfattribs={'color': 2},
)
msp.add_line(
    start=center,
```

(continues on next page)

(continued from previous page)

```
end=ucs.to_wcs(Vector.from_deg_angle(270)),  
dxfattribs={'color': 2},  
)
```



4.3 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.

4.3.1 Drawing

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

Drawing Management

Create New Drawings

`ezdxf.new(dxversion='AC1009')`

Create a new drawing from a template-drawing. The template-drawings are located in a template directory, which resides by default in the *ezdxf* package subfolder *templates*. The location of the template directory can be changed by the global option `ezdxf.options.template_dir`. *dxversion* can be either 'AC1009' the official DXF version name or 'R12' the AutoCAD release name (release name works since *ezdxf* 0.7.4). You can only create new drawings for the following DXF versions:

Version	AutoCAD Release
AC1009	AutoCAD R12
AC1015	AutoCAD R2000
AC1018	AutoCAD R2004
AC1021	AutoCAD R2007
AC1024	AutoCAD R2010
AC1027	AutoCAD R2013
AC1032	AutoCAD R2018

Open Drawings

You can open DXF drawings from disk or from a text-stream. (byte-stream usage is not implemented yet).

`ezdxf.readfile(filename, encoding='auto', legacy_mode=False)`

This is the preferred method to open existing DXF files. Read the DXF drawing from the file-system with auto-detection of encoding. Decoding errors will be ignored. Override encoding detection by setting parameter *encoding* to the estimated encoding. (use Python encoding names like in the `open()` function).

If parameter *legacy_mode* is *True*, *ezdxf* tries to reorder the coordinates of the LINE entity in DXF 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 option *legacy_mode=True* if error “Missing required y coordinate near line: ...” occurs.

`ezdxf.read(stream, legacy_mode=False)`

Read DXF drawing from a text-stream, returns a *Drawing* object. Open the stream in text mode (*mode='rt'*) and the correct encoding has to be set at the open function (in Python 2.7 use `io.open()`), the stream requires at least a `readline()` method. Since DXF version R2007 (AC1021) file encoding is always 'utf-8'.

If parameter *legacy_mode* is *True*, ezdxf tries to reorder the coordinates of the LINE entity in DXF files from CAD applications which wrote the coordinates in the order: x1, x2, y1, y2, see also *readfile()* method.

Save Drawings

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

Global Options

Global options stored in `ezdxf.options`

`ezdxf.options.compress_binary_data`

If you don't need access to binary data of DXF entities, you can compress them in memory for a lower memory footprint, set the global `ezdxf.options.compress_binary_data = True` to compress binary data for every drawing you open, but data compression costs time, so this option isn't active by default. You can individually compress the binary data of a drawing with the method *Drawing.compress_binary_data()*.

`ezdxf.options.template_dir`

Directory where the *new()* function looks for its template file (AC1009.dxf, AC1015.dxf, ...) , default is *None*, which means the package subfolder *templates*. But if you want to use your own templates set this option `ezdxf.options.template_dir = "my_template_directory"`. But you don't really need this, just open your template file with *ezdxf.readfile()* and save the drawing as new file with the *Drawing.saveas()* method.

This option is very useful if the *ezdxf* package resides in a zip archive.

`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 is *True*.

Drawing Object

`class Drawing`

The *Drawing* class manages all entities and tables related to a DXF drawing. Every drawing has its own character *encoding* which is only important for saving to disk.

Drawing Attributes

`Drawing.dxfversion`

contains the DXF version as string like 'AC1009', set by the *new()* or the *readfile()* function. (read only)

`Drawing.acad_version`

contains the AutoCAD release number string like 'R12' or 'R2000' that introduced the DXF version of this drawing. (read only)

`Drawing.encoding`

DXF drawing text encoding, the default encoding for new drawings is 'cp1252'. Starting with DXF version R2007 (AC1021) DXF files are written as UTF-8 encoded text files, regardless of the attribute *Drawing.encoding* (read/write) see also: *DXF File Encoding*

supported	encodings
'cp874'	Thai
'cp932'	Japanese
'gbk'	UnifiedChinese
'cp949'	Korean
'cp950'	TradChinese
'cp1250'	CentralEurope
'cp1251'	Cyrillic
'cp1252'	WesternEurope
'cp1253'	Greek
'cp1254'	Turkish
'cp1255'	Hebrew
'cp1256'	Arabic
'cp1257'	Baltic
'cp1258'	Vietnam

Drawing.filename

Contains the drawing filename, if the drawing was opened with the `readfile()` function else set to *None*. (read/write)

Drawing.dxfactory

DXF entity creation factory, see also `DXFFactory` (read only).

Drawing.sections

Collection of all existing sections of a DXF drawing.

Drawing.header

Shortcut for `Drawing.sections.header`

Reference to the *HeaderSection* of the drawing, where you can change the drawing settings.

Drawing.entities

Shortcut for `Drawing.sections.entities`

Reference to the *EntitySection* of the drawing, where all graphical entities are stored, but only from model space and the *active* layout (paper space). Just for your information: Entities of other layouts are stored as blocks in the *BlocksSection*.

Drawing.blocks

Shortcut for `Drawing.sections.blocks`

Reference to the blocks section, see also *BlocksSection*.

Drawing.layouts

Reference to the layout management object, see also *Layouts*.

Drawing.groups

requires DXF version R13 or later

Table (dict) of all groups used in this drawing, see also *DXFGroupTable*.

Drawing.layers

Shortcut for `Drawing.sections.tables.layers`

Reference to the layers table, where you can create, get and remove layers, see also *Table* and *Layer*

Drawing.styles

Shortcut for `Drawing.sections.tables.styles`

Reference to the styles table, see also *Style*.

Drawing.dimstyles

Shortcut for `Drawing.sections.tables.dimstyles`

Reference to the dimstyles table, see also *DimStyle*.

Drawing.linetypes

Shortcut for `Drawing.sections.tables.linetypes`

Reference to the linetypes table, see also *Linetype*.

Drawing.views

Shortcut for `Drawing.sections.tables.views`

Reference to the views table, see also *View*.

Drawing.viewports

Shortcut for `Drawing.sections.tables.viewports`

Reference to the viewports table, see also *Viewport*.

Drawing.ucs

Shortcut for `Drawing.sections.tables.ucs`

Reference to the ucs table, see also *UCS*.

Drawing.appids

Shortcut for `Drawing.sections.tables.appids`

Reference to the appids table, see also *AppID*.

Drawing.is_binary_data_compressed

Indicates if binary data is compressed in memory. see: *Drawing.compress_binary_data()*

Drawing Methods**Drawing.modelspace()**

Get the model space layout, see also *Layout*.

Drawing.layout(name)

Get a paper space layout by *name*, see also *Layout*. (DXF version AC1009, supports only one paper space layout, so *name* is ignored)

Drawing.layout_names()

Get a list of available paper space layouts.

Drawing.new_layout(name, dxfattribs=None)

Create a new paper space layout *name*. Returns a *Layout* object. Available only for DXF version AC1015 or newer, AC1009 supports only one paper space.

Drawing.delete_layout(name)

Delete paper space layout *name* and all its entities. Available only for DXF version AC1015 or newer, AC1009 supports only one paper space and you can't delete it.

Drawing.layouts_and_blocks()

Iterate over all layouts (mode space and paper space) and all block definitions.

Drawing.chain_layouts_and_blocks()

Chain entity spaces of all layouts and blocks. Yields an iterator for all entities in all layouts and blocks.

Drawing.add_image_def(filename, size_in_pixel, name=None)

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, see *Tutorial for Image and ImageDef*. *name* is the internal image name, if set to None, name is auto-generated.

Parameters

- **filename** – image file name
- **size_in_pixel** – image size in pixel as (x, y) tuple
- **name** – image name for internal use, None for an auto-generated name

Drawing.**add_underlay_def** (*filename, format='pdf', name=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). Returns a UnderlayDef entity which is needed to create an underlay reference, see *Tutorial for Underlay and UnderlayDefinition*. *name* defines the page/sheet to display.

Parameters

- **filename** – underlay file name
- **format** – file format (pdf, dwf or dgn) or ext=get format from filename extension
- **name** – pdf: page number to display; dgn: 'default'; dwf: ????

Drawing.**add_xref_def** (*filename, name*)

Add an external reference (xref) definition to the blocks section.

Add xref to a layout by *Layout.add_blockref()*.

Parameters

- **filename** – external reference filename
- **name** – block name for the xref

Drawing.**save** (*encoding='auto'*)

Write drawing to file-system by using the *filename* attribute as filename. Overwrite 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 file encoding

Drawing.**saveas** (*filename, encoding='auto'*)

Write drawing to file-system by setting the *filename* attribute to *filename*. For argument *encoding* see: *save()*.

Parameters

- **filename** – file name
- **encoding** – override file encoding

Drawing.**write** (*stream*)

Write drawing to a text stream. For DXF version R2004 (AC1018) and prior opened stream with *encoding=Drawing.encoding* and *mode='wt'*. For DXF version R2007 (AC1021) and later use *encoding='utf-8'*.

Drawing.**cleanup** (*groups=True*)

Cleanup drawing. Call it before saving the drawing but only if necessary, the process could take a while.

Parameters groups – removes deleted and invalid entities from groups

Drawing.**compress_binary_data** ()

If you don't need access to binary data of DXF entities, you can compress them in memory for a lower memory footprint, you can set *ezdxf.options.compress_binray_data = True* to compress binary data for every drawing you open, but data compression cost time, so this option isn't active by default.

Drawing.**query** (*query='*'*)

Entity query over all layouts and blocks. (see also: *Name Query String* and *Retrieve Entities by Query Language*)

Excluding the OBJECTS section!

Parameters **query** – query string

Returns *EntityQuery* container

Drawing.**groupby** (*dxfattrib=""*, *key=None*)

Groups DXF entities of all layouts and blocks by an DXF attribute or a key function. (see also: *Retrieve Entities by groupby*)

Excluding the OBJECTS section!

Parameters

- **dxfattrib** – grouping DXF attribute 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

Returns dict

Low Level Access to DXF entities

Drawing.**get_dxf_entity** (*handle*)

Get entity by *handle* from entity database. Low level access to DXF entities database. Raises DXFKeyError if *handle* doesn't exist. Returns DXFEntity or inherited.

If you just need the raw DXF tags use:

```
tags = Drawing.entitydb[handle] # raises DXFKeyError, if handle does not exist
tags = Drawing.entitydb.get(handle) # returns a default value, if handle does not_
↳exist (None by default)
```

type of tags: ExtendedTags

Drawing Header Section

The drawing settings are stored in the header section, which is accessible by the *header* attribute. See the online documentation from Autodesk for available *header variables*.

class HeaderSection

HeaderSection.**__getitem__** (*key*)

Get drawing settings by index operator like: `drawing.header['$ACADVER']`

HeaderSection.**__setitem__** (*key, value*)

Set drawing settings by index operator like: `drawing.header['$ANGDIR'] = 1 # Clockwise angles`

HeaderSection.**custom_vars**

Stores the custom drawing properties in *CustomVars* object.

class CustomVars

Stores custom properties in the DXF header as `$CUSTOMPROPERTYTAG/$CUSTOMPROPERTY` values. Custom properties are just supported at DXF version AC1018 (AutoCAD 2004) or newer. With ezdxf you can create custom properties on older DXF versions, but AutoCAD will not show this properties.

CustomVars.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*).

CustomVars.__len__()
 Count of custom properties.

CustomVars.__iter__()
 Iterate over all custom properties as (*tag*, *value*) tuples.

CustomVars.clear()
 Removes all custom properties.

CustomVars.get(*tag*, *default=None*)
 Returns the value of the first custom property *tag*.

CustomVars.has_tag(*tag*)
 True if custom property *tag* exists, else False.

CustomVars.append(*tag*, *value*)
 Add custom property as (*tag*, *value*) tuple.

CustomVars.replace(*tag*, *value*)
 Replaces the value of the first custom property *tag* by a new *value*. Raises DXFValueError if *tag* does not exist.

CustomVars.remove(*tag*, *all=False*)
 Removes the first occurrence of custom property *tag*, removes all occurrences if *all* is True. Raises DXFValueError if *tag* does not exist.

4.3.2 Tables

Table Class

Generic Table Class

class Table

Table entry names are case insensitive: 'Test' == 'TEST'.

Table.new(*name*, *dxfattribs=None*)

param str name name of the new table-entry

param dict dxfattribs optional table-parameters, these parameters are described at the table-entry-classes below.

returns table-entry-class, can be ignored

Table entry creation is for all tables the same procedure:

```
drawing.tablename.new(name, dxfattribs)
```

Where *tablename* can be: *layers*, *styles*, *linetypes*, *views*, *viewports* or *dimstyles*.

Table.get(*name*)

Get table-entry *name*. Raises DXFValueError if table-entry is not present.

Table.remove(*name*)

Removes table-entry *name*. Raises `DXFValueError` if table-entry is not present.

`Table.__len__()`

Get count of table-entries.

`Table.has_entry(name)`

True if table contains a table-entry *name*.

`Table.__contains__(name)`

True if table contains a table-entry *name*.

`Table.__iter__()`

Iterate over all `table.entries`, yields table-entry-objects.

Style Table Class

class `StyleTable` (*Table*)

`StyleTable.get_shx(name)`

Get existing shx entry, or create a new entry.

`StyleTable.find_shx(name)`

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.

Viewport Table Class

class `ViewportTable` (*Table*)

The viewport table stores the model space 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”.

`ViewportTable.get_config(name)`

Returns a list of `Viewport` objects, of the multi-viewport configuration *name*.

`ViewportTable.delete_config(name)` :

Delete all `Viewport` objects of the multi-viewport configuration *name*.

Layer

class `Layer`

Layer definition, defines attribute values for entities on this layer for their attributes set to `BYLAYER`.

DXF Attributes for Layer

`Layer.dxf.handle`

DXF handle (feature for experts)

`Layer.dxf.owner`

requires DXF R13 or later

`Layer.dxf.name`

Layer name (str)

`Layer.dxf.flags`

Layer flags (feature for experts)

`Layer.dxf.color`

Layer color, but use `Layer.get_color()`, because color is negative for layer status *off* (int)

`Layer.dxf.linetype`

Name of line type (str)

`Layer.dxf.plot`

Plot flag (int)

- 1 = plot layer (default value)
- 0 = don't plot layer

`Layer.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.

requires DXF R13 or later

`Layer.dxf.plot_style_name`

Handle to PlotStyleName (feature for experts)

requires DXF R13 or later

`Layer.dxf.line_weight`

requires DXF R13 or later

`Layer.dxf.plot_style_name`

requires DXF R13 or later

`Layer.dxf.material`

requires DXF R13 or later

Layer Methods

`Layer.is_frozen()`

`Layer.freeze()`

`Layer.thaw()`

`Layer.is_locked()`

`Layer.lock()`

Lock layer, entities on this layer are not editable - just important in CAD applications.

`Layer.unlock()`

Unlock layer, entities on this layer are editable - just important in CAD applications.

`Layer.is_off()`

`Layer.is_on()`

`Layer.on()`

Switch layer *on* (visible).

`Layer.off()`

Switch layer *off* (invisible).

`Layer.get_color()`

Get layer color, preferred method for getting the layer color, because color is negative for layer status *off*.

`Layer.set_color(color)`

Set layer color to *color*, preferred method for setting the layer color, because color is negative for layer status *off*.

Style

class Style

Defines a text style, can be used by entities: *Text*, *Attrib* and *Attdef*

DXF Attributes for Style

`Style.dxf.handle`

DXF handle (feature for experts)

`Style.dxf.owner`

requires DXF R13 or later

`Style.dxf.name`

Style name (str)

`Style.dxf.flags`

Style flags (feature for experts)

`Style.dxf.height`

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

`Style.dxf.width`

Width factor (float), default is 1

`Style.dxf.oblique`

Oblique angle in degrees, 0 is vertical (float)

`Style.dxf.text_generation_flags`

Text generations flags (int)

- 2 = text is backward (mirrored in X)
- 4 = text is upside down (mirrored in Y)

`Style.dxf.last_height`

Last height used in drawing units (float)

`Style.dxf.font`

Primary font file name (str)

`Style.dxf.bigfont`

Big font name, blank if none (str)

Linetype

class Linetype

Defines a linetype.

DXF Attributes for Linetype

`Linetype.dxf.name`

Linetype name (str)

`Linetype.dxf.owner`

requires DXF R13 or later

`Linetype.dxf.description`

Linetype description (str)

`Linetype.dxf.length`

Total pattern length in drawing units (float)

`Linetype.dxf.items`

Number of linetype elements (int)

See also:

[Tutorial for Linetypes](#)

DXF Internals: *[LTYPE Table](#)*

DimStyle

class DimStyle

Defines a dimension style.

DXF Attributes for DimStyle

`DimStyle.dxf.handle`

Handle of table entry.

`DimStyle.dxf.owner`

Handle to dimstyle table, requires DXF R13 or later

`DimStyle.dxf.name`

Text style name.

`DimStyle.dxf.flags`

Standard flag values (bit-coded values):

- 16 = If set, table entry is externally dependent on an xref
- 32 = If both this bit and bit 16 are set, the externally dependent xref has been successfully resolved
- 64 = If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is 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)

`DimStyle.dxf.dimpost`

Prefix/suffix for primary units dimension values.

`DimStyle.dxf.dimapost`

Prefix/suffix for alternate units dimensions.

`DimStyle.dxf.dimblk`

Block type to use for both arrowheads. (R2000 obsolete, now object ID)

`DimStyle.dxf.dimblk1` (*R2000 obsolete, now object ID*)

Block type to use for first arrowhead.

`DimStyle.dxf.dimblk2` (*R2000 obsolete, now object ID*)

Block type to use for second arrowhead.

`DimStyle.dxf.dimscale`

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

`DimStyle.dxf.dimasz`

Dimension line and arrowhead size. (default=0.28)

`DimStyle.dxf.dimexo`

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

`DimStyle.dxf.dimdli`

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

`DimStyle.dxf.dimexe`

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

`DimStyle.dxf.dimrnd`

Rounding value for dimensions. (default=0)

`DimStyle.dxf.dimdle`

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

`DimStyle.dxf.dimtp`

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

`DimStyle.dxf.dimtm`

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

`DimStyle.dxf.dimtxt`

Size of dimension text. (default imperial=0.28, default metric=2.5)

`DimStyle.dxf.dimcen`

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

`DimStyle.dxf.dimtsz`

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

`DimStyle.dxf.dimaltf`

Alternate units dimension scale factor. (default=25.4)

`DimStyle.dxf.dimlfac`

Scale factor for linear dimension values. (default=1)

`DimStyle.dxf.dimtvp`

Vertical position of text above or below dimension line. (default=0)

`DimStyle.dxf.dimtfac`

Scale factor for fractional or tolerance text size. (default=1)

`DimStyle.dxf.dimgap`

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

`DimStyle.dxf.dimaltrnd`

Rounding value for alternate dimension units. (default=0)

`DimStyle.dxf.dimtol`

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

`DimStyle.dxf.dimlim`

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

`DimStyle.dxf.dimtih`

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

`DimStyle.dxf.dimtoh`

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

`DimStyle.dxf.dimse1`

Toggles suppression of first extension line. (default=0)

`DimStyle.dxf.dimse2`

Toggles suppression of second extension line. (default=0)

`DimStyle.dxf.dimtad`

Sets text placement relative to dimension line. (default imperial=0, default metric=1)

`DimStyle.dxf.dimzin`

Zero suppression for primary units dimensions. (default imperial=0, default metric=8) ???

`DimStyle.dxf.dimazin`

Controls zero suppression for angular dimensions. (default=0)

`DimStyle.dxf.dimalt`

Enables or disables alternate units dimensioning. (default=0)

`DimStyle.dxf.dimaltd`

Controls decimal places for alternate units dimensions. (default imperial=2, default metric=3)

`DimStyle.dxf.dimtofl`

Toggles forced dimension line creation. (default imperial=0, default metric=1)

`DimStyle.dxf.dimsah`

Toggles appearance of arrowhead blocks. (default=0)

`DimStyle.dxf.dimtix`

Toggles forced placement of text between extension lines. (default=0)

`DimStyle.dxf.dimsoxd`

Suppresses dimension lines outside extension lines. (default=0)

`DimStyle.dxf.dimclrd`

Dimension line, arrowhead, and leader line color. (default=0)

`DimStyle.dxf.dimclre`

Dimension extension line color. (default=0)

`DimStyle.dxf.dimclrt`

Dimension text color. (default=0)

`DimStyle.dxf.dimadec`

Controls the number of decimal places for angular dimensions.

`DimStyle.dxf.dimunit`

Obsolete, now use DIMLUNIT AND DIMFRAC

`DimStyle.dxf.dimdec`

Decimal places for dimension values. (default imperial=4, default metric=2)

`DimStyle.dxf.dimtdec`

Decimal places for primary units tolerance values. (default imperial=4, default metric=2)

`DimStyle.dxf.dimaltu`

Units format for alternate units dimensions. (default=2)

`DimStyle.dxf.dimaltttd`

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

`DimStyle.dxf.dimaunit`

Unit format for angular dimension values. (default=0)

`DimStyle.dxf.dimfrac`

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

`DimStyle.dxf.dimlunit`

Specifies units for all nonangular dimensions. (default=2)

`DimStyle.dxf.dimdsep`

Specifies a single character to use as a decimal separator. (default imperial=".", default metric="")

`DimStyle.dxf.dimtmove`

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

`DimStyle.dxf.dimjust`

Horizontal justification of dimension text. (default=0)

`DimStyle.dxf.dimsd1`

Toggles suppression of first dimension line. (default=0)

`DimStyle.dxf.dimsd2`

Toggles suppression of second dimension line. (default=0)

`DimStyle.dxf.dimtolj`

Vertical justification for dimension tolerance text. (default=1)

`DimStyle.dxf.dimaltz`

Zero suppression for alternate units dimension values. (default=0)

`DimStyle.dxf.dimalttz`

Zero suppression for alternate units tolerance values. (default=0)

`DimStyle.dxf.dimfit`

Obsolete, now use DIMATFIT and DIMTMOVE

`DimStyle.dxf.dimupt`

Controls user placement of dimension line and text. (default=0)

`DimStyle.dxf.dimatfit`

Controls placement of text and arrowheads when there is insufficient space between the extension lines. (default=3)

`DimStyle.dxf.dimtxsty_handle`

Text style used for dimension text, handle of referenced style.

`DimStyle.dxf.dimldrblk_handle`

Controls the type of arrowhead used for leaders, handle of referenced block.

`DimStyle.dxf.dimblk_handle`

Block type to use for both arrowheads, handle of referenced block.

`DimStyle.dxf.dimblk1_handle`

Block type to use for first arrowhead, handle of referenced block.

`DimStyle.dxf.dimblk2_handle`

Block type to use for second arrowhead, handle of referenced block.

`DimStyle.dxf.dimlwd`

Lineweight value for dimension lines. (default=-2, BYBLOCK)

`DimStyle.dxf.dimlwe`

Lineweight value for extension lines. (default=-2, BYBLOCK)

VPort

The viewport table stores the model space viewport configurations. So this entries just model space viewports, not paper space viewports, for paper space viewports see the `Viewport` entity.

class VPort

Defines a viewport to the model space.

DXF Attributes for VPort

`VPort.dxf.handle`

`VPort.dxf.owner`

requires DXF R13 or later

`VPort.dxf.name`

`VPort.dxf.flags`

`VPort.dxf.lower_left`

`VPort.dxf.upper_right`

`VPort.dxf.center_point`

`VPort.dxf.snap_base`

`VPort.dxf.snap_spacing`

`VPort.dxf.grid_spacing`

`VPort.dxf.direction_point`

`VPort.dxf.target_point`

`VPort.dxf.height`

`VPort.dxf.aspect_ratio`

`VPort.dxf.lens_length`

`VPort.dxf.front_clipping`

`VPort.dxf.back_clipping`

`VPort.dxf.snap_rotation`

`VPort.dxf.view_twist`

`VPort.dxf.status`

`VPort.dxf.view_mode`

`VPort.dxf.circle_zoom`

`VPort.dxf.fast_zoom`

`VPort.dxf.ucs_icon`

`VPort.dxf.snap_on`

`VPort.dxf.grid_on`

`VPort.dxf.snap_style`

`VPort.dxf.snap_isopair`

See also:

DXF Internals: *VPORT Configuration Table*

View

The View table stores named views of the model or paper space 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.

class View

DXF Attributes for View

View.dxf.**handle**

View.dxf.**owner**

requires DXF R13 or later

View.dxf.**name**

View.dxf.**flags**

View.dxf.**height**

View.dxf.**width**

View.dxf.**center_point**

View.dxf.**direction_point**

View.dxf.**target_point**

View.dxf.**lens_length**

View.dxf.**front_clipping**

View.dxf.**back_clipping**

View.dxf.**view_twist**

View.dxf.**view_mode**

See also:

DXF Internals: *VIEW Table*

AppID

class AppID

Defines an AppID.

DXF Attributes for AppID

AppID.dxf.**handle**

AppID.dxf.**owner**

requires DXF R13 or later

`AppID.dxf.name`

`AppID.dxf.flags`

UCS

class UCS

Defines an user coordinate system (UCS).

DXF Attributes for UCS

`UCS.dxf.handle`

`UCS.dxf.owner`

requires DXF R13 or later

`UCS.dxf.name`

`UCS.dxf.flags`

`UCS.dxf.origin`

`UCS.dxf.xaxis`

`UCS.dxf.yaxis`

UCS Methods

`UCS.ucs()`

Returns an *ezdxf.algebra.UCS* object for this UCS table entry.

BlockRecord

class BlockRecord

Defines a BlockRecord, exist just in DXF version R13 and later.

DXF Attributes for BlockRecord

`BlockRecord.dxf.handle`

`BlockRecord.dxf.owner`

requires DXF R13 or later

`BlockRecord.dxf.name`

`BlockRecord.dxf.layout`

4.3.3 Layouts

Layout Manager

The layout manager is unique to each DXF drawing, access the layout manager by *Drawing.layouts*.

class `Layouts`

The *Layouts* class manages paper space layouts and the model space.

`Layouts.__len__()`

Return the count for layouts.

`Layouts.__contains__(name)`

Support for the `in` operator

Parameters `name` (*str*) – layout name as shown in tab

`Layouts.__iter__()`

Iterate over model space layout and all paper space layouts as *Layout* objects.

`Layouts.modelspace()`

Returns the model space layout as *Layout* object.

`Layouts.names()`

Returns iterable of all layout names.

`Layouts.get(name)`

Returns layout *name* as *Layout* object.

Parameters `name` (*str*) – layout name as shown in tab

`Layouts.rename(old_name, new_name)`

Rename a layout. Layout *Model* can not be renamed and the new name of a layout must not exist.

Parameters

- `old_name` (*str*) – actual layout name
- `new_name` (*str*) – new layout name

`Layouts.names_in_taborder()`

Returns all layout names in tab order as a list of strings.

`Layouts.new(name, dxfattribs=None)`

Create a new *Layout*.

Parameters `name` (*str*) – layout name as shown in tab

`Layouts.delete(name)`

Delete layout and all entities in this layout.

Parameters `name` (*str*) – layout name as shown in tab

Layout

A *Layout* represents and manages drawing entities, there are three different layout objects:

- Model space is the common working space, containing basic drawing entities.
- Paper spaces are arrangements of objects for printing and plotting, this layout contains basic drawing entities and viewports to the model-space.

- `BlockLayout` works on an associated `Block`, Blocks are collections of drawing entities for reusing by block references.

class `Layout`

Paper Space Layout Setup

`Layout.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). DXF R12 not supported yet.

Parameters

- **size** – paper size as (width, height) tuple
- **margins** – (top, right, bottom, left) hint: clockwise
- **units** – ‘mm’ or ‘inch’
- **offset** – plot origin offset as (x, y) tuple
- **rotation** – 0=no rotation, 1=90deg count-clockwise, 2=upside-down, 3=90deg clockwise
- **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

`Layout.reset_viewports` ()

Delete all existing viewports, and add a new main viewport. (in `page_setup()` included)

`Layout.reset_extends` ()

Reset paper space extends. (in `page_setup()` included)

`Layout.reset_paper_limits` ()

Reset paper space limits. (in `page_setup()` included)

`Layout.get_paper_limits` ()

Returns paper limits in plot paper units, relative to the plot origin, as tuple ((x1, y1), (x2, y2)). Lower left corner is (x1, y1), upper right corner is (x2, y2).

The plot origin is lower left corner of printable area + plot origin offset.

`Layout.set_plot_type` (*value*=5)

Set plot type:

- 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_limits()`)
- 5 = layout information (default)

`Layout.set_plot_style` (*name*='ezdxf.ctb', *show*=False)

Set current plot style e.g. “acad.ctb”, and *show* impact of plot style also on screen.

`Layout.set_plot_window` (*lower_left=(0, 0), upper_right=(0, 0)*)
 Set plot window size in (scaled) paper space units, and relative to the plot origin.

Access Existing Entities

`Layout.__iter__` ()
 Iterate over all drawing entities in this layout.

`Layout.__contains__` (*entity*)
 Test if the layout contains the drawing element *entity* (aka *in* operator).

`Layout.query` (*query='*'*)
 Get included DXF entities matching the *Entity Query String query*. Returns a sequence of type *EntityQuery*.

`Layout.groupby` (*dxfattrib=""*, *key=None*)
 Returns a dict of entity lists, where entities are grouped by a *dxfattrib* or a key function.

Parameters

- **dxfattrib** (*str*) – grouping DXF attribute like ‘layer’
- **key** (*function*) – 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

Create New Entities

`Layout.add_point` (*location*, *dxfattribs=None*)
 Add a *Point* element at *location*.

`Layout.add_line` (*start*, *end*, *dxfattribs=None*)
 Add a *Line* element, starting at 2D/3D point *start* and ending at the 2D/3D point *end*.

`Layout.add_circle` (*center*, *radius*, *dxfattribs=None*)
 Add a *Circle* element, *center* is 2D/3D point, *radius* in drawing units.

`Layout.add_ellipse` (*center*, *major_axis=(1, 0, 0)*, *ratio=1*, *start_param=0*, *end_param=6.283185307*, *dxfattribs=None*)
 Add an *Ellipse* element, *center* is 2D/3D point, *major_axis* as vector, *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.

`Layout.add_arc` (*center*, *radius*, *start_angle*, *end_angle*, *is_counter_clockwise=True*, *dxfattribs=None*)
 Add an *Arc* element, *center* is 2D/3D point, *radius* in drawing units, *start_angle* and *end_angle* in degrees. 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.

`Layout.add_solid` (*points*, *dxfattribs=None*)
 Add a *Solid* element, *points* is list of 3 or 4 2D/3D points.

`Layout.add_trace` (*points*, *dxfattribs=None*)
 Add a *Trace* element, *points* is list of 3 or 4 2D/3D points.

`Layout.add_3dface` (*points*, *dxfattribs=None*)
 Add a *3DFace* element, *points* is list of 3 or 4 2D/3D points.

`Layout.add_text` (*text*, *dxfattribs=None*)
 Add a *Text* element, *text* is a string, see also *Style*.

- Layout.**add_blockref** (*name, insert, dxfattribs=None*)
Add an *Insert* element, *name* is the block name, *insert* is a 2D/3D point.
- Layout.**add_auto_blockref** (*name, insert, values, dxfattribs=None*)
Add an *Insert* element, *name* is the block name, *insert* is a 2D/3D point. Add *Attdef*, defined in the block definition, automatically as *Attrib* to the block reference, and set text of *Attrib*. *values* is a dict with key=tag, value=text values. The *Attrib* elements are placed relative to the insert point = block base point.
- Layout.**add_attrib** (*tag, text, insert, dxfattribs=None*)
Add an *Attrib* element, *tag* is the attrib-tag, *text* is the attrib content.
- Layout.**add_polyline2d** (*points, dxfattribs=None*)
Add a *Polyline* element, *points* is list of 2D points.
- Layout.**add_polyline3d** (*points, dxfattribs=None*)
Add a *Polyline* element, *points* is list of 3D points.
- Layout.**add_polymesh** (*size=(3, 3), dxfattribs=None*)
Add a *Polymesh* element, *size* is a 2-tuple (*mcount, ncount*). A polymesh is a grid of *mcount* x *ncount* vertices and every vertex has its own xyz-coordinates.
- Layout.**add_polyface** (*dxfattribs=None*)
Add a *Polyface* element.
- Layout.**add_lwpolyline** (*points, dxfattribs=None*)
Add a 2D polyline, *points* is a list of (x, y, [start_width, [end_width, [bulge]]]) tuples. Set start_width, end_width to 0 to be ignored (x, y, 0, 0, bulge). A *LWPolyline* is defined as a single graphic entity and consume less disk space and memory. (requires DXF version AC1015 or later)
- Layout.**add_mtext** (*text, dxfattribs=None*)
Add a *MText* element, which is a multiline text element with automatic text wrapping at boundaries. The *char_height* is the initial character height in drawing units, *width* is the width of the text boundary in drawing units. (requires DXF version AC1015 or later)
- Layout.**add_shape** (*name, insert=(0, 0, 0), size=1.0, dxfattribs=None*)
Add a *Shape* reference to a external stored shape.
- Layout.**add_ray** (*start, unit_vector, dxfattribs=None*)
Add a *Ray* that starts at a point and continues to infinity (construction line). (requires DXF version AC1015 or later)
- Layout.**add_xline** (*start, unit_vector, dxfattribs=None*)
Add an infinity *XLine* (construction line). (requires DXF version AC1015 or later)
- Layout.**add_spline** (*fit_points=None, dxfattribs=None*)
Add a *Spline*, *fit_points* has to be a list (container or generator) of (x, y, z) tuples. (requires DXF version AC1015 or later)

AutoCAD creates a spline through fit points by a proprietary algorithm. *ezdxf* can not reproduce the control point calculation.
- Layout.**add_open_spline** (*control_points, degree=3, dxfattribs=None*)
Add an open uniform *Spline*, *control_points* has to be a list (container or generator) of (x, y, z) tuples, *degree* specifies degree of spline. (requires DXF version AC1015 or later)

Open uniform B-splines start and end at your first and last control points.
- Layout.**add_closed_spline** (*control_points, degree=3, dxfattribs=None*)
Add a closed uniform *Spline*, *control_points* has to be a list (container or generator) of (x, y, z) tuples, *degree* specifies degree of spline. (requires DXF version AC1015 or later)

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

Layout.**add_rational_spline** (*control_points, weights, degree=3, dxfattribs=None*)

Add an open rational uniform *Spline*, *control_points* has to be a list (container or generator) of (x, y, z) tuples, *weights* has to be a list of values, which defines the influence of the associated control point, therefore count of control points has to be equal to the count of weights, *degree* specifies degree of spline. (requires DXF version AC1015 or later)

Open rational uniform B-splines start and end at your first and last control points, and have additional control possibilities by weighting each control point.

Layout.**add_closed_rational_spline** (*control_points, weights, degree=3, dxfattribs=None*)

Add a closed rational uniform *Spline*, *control_points* has to be a list (container or generator) of (x, y, z) tuples, *weights* has to be a list of values, which defines the influence of the associated control point, therefore count of control points has to be equal to the count of weights, *degree* specifies degree of spline. (requires DXF version AC1015 or later)

Closed rational uniform B-splines start and end at the first control point, and have additional control possibilities by weighting each control point.

Layout.**add_spline_control_frame** (*fit_points, degree=3, method='distance', power=.5, dxfattribs=None*)

Create and add B-spline control frame from fit points.

Supported methods are:

- **uniform**: creates a uniform t vector, [0 .. 1] equally spaced
- **distance**: creates a t vector with values proportional to the fit point distances
- **centripetal**: creates a t vector with values proportional to the fit point distances^{power}

None of this methods matches the spline created from fit points by AutoCAD.

Parameters

- **fit_points** – fit points of B-spline
- **degree** – degree of B-spline
- **method** – calculation method for parameter vector t
- **power** – power for centripetal method
- **dxfattribs** – DXF attributes for SPLINE entity

Returns DXF *Spline* object

Layout.**add_spline_approx** (*fit_points, count, degree=3, method='distance', power=.5, dxfattribs=None*)

Approximate B-spline by a reduced count of control points, given are the fit points and the degree of the B-spline.

- **uniform**: creates a uniform t vector, [0 .. 1] equally spaced
- **distance**: creates a t vector with values proportional to the fit point distances
- **centripetal**: creates a t vector with values proportional to the fit point distances^{power}

Parameters

- **fit_points** – all fit points of B-spline
- **count** – count of designated control points
- **degree** – degree of B-spline
- **method** – calculation method for parameter vector t
- **power** – power for centripetal method

- **dxfattribs** – DXF attributes for SPLINE entity

Returns DXF *Spline* object

Layout.**add_body** (*acis_data=""*, *dxfattribs=None*)

Add a *Body* entity, *acis_data* has to be a list (container or generator) of text lines **without** line endings. (requires DXF version AC1015 or later)

Layout.**add_region** (*acis_data=""*, *dxfattribs=None*)

Add a *Region* entity, *acis_data* has to be a list (container or generator) of text lines **without** line endings. (requires DXF version AC1015 or later)

Layout.**add_3dsolid** (*acis_data=""*, *dxfattribs=None*)

Add a *3DSolid* entity, *acis_data* has to be a list (container or generator) of text lines **without** line endings. (requires DXF version AC1015 or later)

Layout.**add_hatch** (*color=7*, *dxfattribs=None*)

Add a *Hatch* entity, *color* as ACI (AutoCAD Color Index), default is 7 (black/white). (requires DXF version AC1015 or later)

Layout.**add_image** (*image_def*, *insert*, *size_in_units*, *rotation=0*, *dxfattribs=None*)

Add an *Image* entity, *insert* is the insertion point as (x, y [,z]) tuple, *size_in_units* is the image size as (x, y) tuple in drawing units, *image_def* is the required *ImageDef*, *rotation* is the rotation angle around the z-axis in degrees. Create *ImageDef* by the *Drawing* factory function *add_image_def()*, see *Tutorial for Image and ImageDef*. (requires DXF version AC1015 or later)

Layout.**add_underlay** (*underlay_def*, *insert=(0, 0, 0)*, *scale=(1, 1, 1)*, *rotation=0*, *dxfattribs=None*)

Add an *Underlay* entity, *insert* is the insertion point as (x, y [,z]) tuple, *scale* is the underlay scaling factor as (x, y, z) tuple, *underlay_def* is the required *UnderlayDefinition*, *rotation* is the rotation angle around the z-axis in degrees. Create *UnderlayDef* by the *Drawing* factory function *add_underlay_def()*, see *Tutorial for Underlay and UnderlayDefinition*. (requires DXF version AC1015 or later)

Layout.**add_entity** (*dxfentity*)

Add an existing DXF entity to a layout, but be sure to unlink (*unlink_entity()*) first the entity from the previous owner layout.

Change Redraw Order

Layout.**set_redraw_order** (*handles*)

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 this entities, this redefines the order in which the entities are regenerated. Parameter *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 object handles can share the same sort handle, and sort handles can collide with existing handles too. Also 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.

Changing redraw order just works for model space and paper space layouts, not for block layouts.

Parameters handles – list or dict of handle associations

Layout.**get_redraw_order** ()

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

Delete Entities

Layout.**unlink_entity** (*entity*)

Unlink *entity* from layout but does not delete entity from the drawing database.

Layout.**delete_entity** (*entity*)

Delete *entity* from layout and drawing database.

Layout.**delete_all_entities** ()

Delete all *entities* from layout and drawing database.

Model Space

class Modelspace (*Layout*)

At this time the *Modelspace* class is the *Layout* class.

Modelspace.**new_geodata** (*dxfattribs=None*)

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 (AC1024) or later. 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.

Modelspace.**get_geodata** (*dxfattribs=None*)

Returns the *GeoData* entity associated to this layout or None.

Paper Space

class Paperspace (*Layout*)

At this time the *Paperspace* class is the *Layout* class.

BlockLayout

class BlockLayout (*Layout*)

BlockLayout.**name**

The name of the associated block element. (read/write)

BlockLayout.**block**

Get the associated DXF *BLOCK* entity.

BlockLayout.**is_layout_block**

True if block is a model space or paper space block definition.

BlockLayout.**add_attdef** (*tag, insert=(0, 0), dxfattribs=None*)

Add an *Attdef* element, *tag* is the attribute-tag, *insert* is the 2D/3D insertion point of the Attribute. Set position and alignment by the idiom:

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

BlockLayout.**attdefs** ()

Iterator for included *Attdef* entities.

BlockLayout.**has_attdef** (*tag*)

Returns *True* if an attdef *tag* exists else *False*.

`BlockLayout.get_attdef(tag)`

Get the attribute definition object `Attdef` with `object.dxf.tag == tag`, returns `None` if not found.

`BlockLayout.get_attdef_text(tag, default=None)`

Get content text for attdef `tag` as string or return `default` if no attdef `tag` exists.

4.3.4 Entities

Graphic Base Class

class GraphicEntity

Common base class for all graphic entities.

`GraphicEntity.dxf`

(read only) The DXF attributes namespace, access DXF attributes by this attribute, like `entity.dxf.layer = 'MyLayer'`. Just the `dxf` attribute is read only, the DXF attributes are read- and writeable.

`GraphicEntity.drawing`

(read only) Get the associated drawing.

`GraphicEntity.dxfactory`

(read only) Get the associated DXF factory. (feature for experts)

`GraphicEntity.rgb`

(read/write) Get/Set true color as RGB-Tuple. This attribute does not exist in DXF AC1009 (R12) entities, the attribute exists in DXF AC1015 entities but does not work (raises `DXFValueError`), requires at least DXF Version AC1018 (AutoCAD R2004). usage: `entity.rgb = (30, 40, 50)`;

`GraphicEntity.transparency`

(read/write) Get/Set transparency value as float. This attribute does not exist in DXF AC1009 (R12) entities, the attribute exists in DXF AC1015 entities but does not work (raises `DXFValueError`), requires at least DXF Version AC1018 (AutoCAD R2004). Value range 0.0 to 1.0 where 0.0 means entity is opaque and 1.0 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 `entity.dxf.transparency`

`GraphicEntity.dxf_type()`

Get the DXF type string, like `LINE` for the line entity.

`GraphicEntity.copy()`

Deep copy of `DXFEntity` with new handle and duplicated linked entities (`VERTEX`, `ATTRIB`, `SEQEND`). The new entity is not included in any layout space, so the owner tag is set to '0' for undefined owner/layout.

Use `Layout.add_entity()` to add the duplicated entity to a layout, layout can be the model space, a paper space layout or a block layout.

This is not a deep copy in the meaning of Python, because handle, link and owner is changed.

`GraphicEntity.copy_to_layout(layout)`

Duplicate entity and add new entity to `layout`.

`GraphicEntity.move_to_layout(layout, source=None)`

Move entity from actual layout to `layout`. For DXF R12 providing `source` is faster, if the entity resides in a block layout, because `ezdxf` has to search in all block layouts, else `source` is not required.

`GraphicEntity.get_dxf_attrib(key, default=DXFValueError)`

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:

```
layer = entity.get_dxf_attrib("layer")
# same as
layer = entity.dxf.layer
```

`GraphicEntity.set_dxf_attrib(key, value)`

Set DXF attribute *key* to *value*:

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

`GraphicEntity.del_dxf_attrib(key)`

Delete/remove DXF attribute *key*. Raises `AttributeError` if *key* isn't supported.

`GraphicEntity.dxf_attrib_exists(key)`

Returns *True* if DXF attrib *key* really exists else *False*. Raises `AttributeError` if *key* isn't supported

`GraphicEntity.supported_dxf_attrib(key)`

Returns *True* if DXF attrib *key* is supported by this entity else *False*. Does not grant that attrib *key* really exists.

`GraphicEntity.valid_dxf_attrib_names(key)`

Returns a list of supported DXF attribute names.

`GraphicEntity.dxfattribs()`

Create a dict() with all accessible DXF attributes and their value, not all data is accessible by dxf attributes like definition points of *LWPolyline* or *Spline*

`GraphicEntity.update_attribs(dxfattribs)`

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

`GraphicEntity.set_flag_state(flag, state=True, name='flags')`

Set binary coded *flag* of DXF attribute *name* to 1 (on) if *state* is *True*, set *flag* to 0 (off) if *state* is *False*.

`GraphicEntity.get_flag_state(flag, name='flags')`

Returns *True* if any *flag* of DXF attribute is 1 (on), else *False*. Always check just one flag state at the time.

`GraphicEntity.get_layout()`

Returns the *Layout* which contains this entity, *None* if entity is not assigned to any layout.

`GraphicEntity.get_ocs()`

Returns an OCS object, see also: *OCS*

Common DXF Attributes for DXF R12

`GraphicEntity.dxf.handle`

DXF handle (feature for experts)

`GraphicEntity.dxf.layer`

layer name as string; default=0

`GraphicEntity.dxf.linetype`

linetype as string, special names BYLAYER, BYBLOCK; default=BYLAYER

`GraphicEntity.dxf.color`

dxf color index, 0 ... BYBLOCK, 256 ... BYLAYER; default=256

`GraphicEntity.dxf.paperspace`

0 for entity resides in model-space, 1 for paper-space, this attribute is set automatically by adding an entity to a layout (feature for experts); default=0

`GraphicEntity.dxf.extrusion`
 extrusion direction as 3D point; default=(0, 0, 1)

Common DXF Attributes for DXF R13 or later

`GraphicEntity.dxf.handle`
 DXF handle (feature for experts)

`GraphicEntity.dxf.owner`
 handle to owner, it's a BLOCK_RECORD entry (feature for experts)

`GraphicEntity.dxf.layer`
 layer name as string; default = 0

`GraphicEntity.dxf.linetype`
 linetype as string, special names BYLAYER, BYBLOCK; default=BYLAYER

`GraphicEntity.dxf.color`
 dxf color index, default = 256

- 0 = BYBLOCK
- 256 = BYLAYER
- 257 = BYOBJECT

`GraphicEntity.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.

Constants defined in `ezdxf.lldxf.const`

- `LINEWEIGHT_BYLAYER` = -1
- `LINEWEIGHT_BYBLOCK` = -2
- `LINEWEIGHT_DEFAULT` = -3

`GraphicEntity.dxf.ltscale`
 line type scale as float; default=1.0

`GraphicEntity.dxf.invisible`
 1 for invisible, 0 for visible; default=0

`GraphicEntity.dxf.paperspace`
 0 for entity resides in model-space, 1 for paper-space, this attribute is set automatically by adding an entity to a layout (feature for experts); default=0

`GraphicEntity.dxf.extrusion`
 extrusion direction as 3D point; default=(0, 0, 1)

`GraphicEntity.dxf.thickness`
 entity thickness as float; default=0

`GraphicEntity.dxf.true_color`
 true color value as int 0x00RRGGBB, requires DXF Version AC1018 (AutoCAD R2004)

`GraphicEntity.dxf.color_name`
 color name as string (R2004)

`GraphicEntity.dxf.transparency`
 transparency value as int, 0x020000TT 0x00 = 100% transparent / 0xFF = opaque (R2004)

GraphicEntity.dxf.**shadow_mode** (*R2007*)

- 0 = casts and receives shadows
- 1 = casts shadows
- 2 = receives shadows
- 3 = ignores shadows

Point

class Point (*GraphicEntity*)

A point at location *point*, *dxftype* is POINT. Create points in layouts and blocks by factory function `add_point()`.

DXF Attributes for Point

Common DXF Attributes for DXF R12

Common DXF Attributes for DXF R13 or later

Point.dxf.**location**

Location of the point (2D/3D Point in *WCS*)

Line

class Line (*GraphicEntity*)

A line from *start* to *end*, *dxftype* is LINE. Create lines in layouts and blocks by factory function `add_line()`.

DXF Attributes for Line

Common DXF Attributes for DXF R12

Common DXF Attributes for DXF R13 or later

Line.dxf.**start**

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

Line.dxf.**end**

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

Circle

class Circle (*GraphicEntity*)

A circle at location *center* and *radius*, *dxftype* is CIRCLE. Create circles in layouts and blocks by factory function `add_circle()`.

DXF Attributes for Circle

Common DXF Attributes for DXF R12

Common DXF Attributes for DXF R13 or later

Circle.dxf.**center**

center point of circle (2D/3D Point in *OCS*)

Circle.dxf.**radius**

radius of circle (float)

Arc

class Arc (*GraphicEntity*)

An arc at location *center* and *radius* from *start_angle* to *end_angle*, *dxftype* is ARC. The arc goes from *start_angle* to *end_angle* in *counter clockwise* direction. Create arcs in layouts and blocks by factory function `add_arc()`.

DXF Attributes for Arc

Common DXF Attributes for DXF R12

Common DXF Attributes for DXF R13 or later

Arc.dxf.**center**

center point of arc (2D/3D Point in *OCS*)

Arc.dxf.**radius**

radius of arc (float)

Arc.dxf.**start_angle**

start angle in degrees (float)

Arc.dxf.**end_angle**

end angle in degrees (float)

Text

class Text (*GraphicEntity*)

A simple one line text, *dxftype* is TEXT. Text height is in drawing units and defaults to 1, but it depends on the rendering software what you really get. Width is a scaling factor, but it is not defined what is scaled (I assume the text height), but it also depends on the rendering software what you get. This is one reason why DXF and also DWG are not reliable for exchanging exact styling, they are just reliable for exchanging exact geometry. Create text in layouts and blocks by factory function `add_text()`.

DXF Attributes for Text

Common DXF Attributes for DXF R12

Common DXF Attributes for DXF R13 or later

`Text.dxf.text`

the content text itself (str)

`Text.dxf.insert`

first alignment point of text (2D/3D Point in *OCS*), relevant for the adjustments LEFT, ALIGN and FIT.

`Text.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)

`Text.dxf.height`

text height in drawing units (float); default=1

`Text.dxf.rotation`

text rotation in degrees (float); default=0

`Text.dxf.oblique`

text oblique angle (float); default=0

`Text.dxf.style`

text style name (str); default=STANDARD

`Text.dxf.width`

width scale factor (float); default=1

`Text.dxf.halign`

horizontal alignment flag (int), use `Text.set_pos()` and `Text.get_align()`; default=0

`Text.dxf.valign`

vertical alignment flag (int), use `Text.set_pos()` and `Text.get_align()`; default=0

`Text.dxf.text_generation_flag`

text generation flags (int)

- 2 = text is backward (mirrored in X)
- 4 = text is upside down (mirrored in Y)

Text Methods

`Text.set_pos(p1, p2=None, align=None)`

param p1 first alignment point as (x, y[, z])-tuple

param p2 second alignment point as (x, y[, z])-tuple, required for ALIGNED and FIT else ignored

param str align new alignment, None for preserve existing alignment.

Set text alignment, valid positions are:

Vert/Horiz	Left	Center	Right
Top	TOP_LEFT	TOP_CENTER	TOP_RIGHT
Middle	MIDDLE_LEFT	MIDDLE_CENTER	MIDDLE_RIGHT
Bottom	BOTTOM_LEFT	BOTTOM_CENTER	BOTTOM_RIGHT
Baseline	LEFT	CENTER	RIGHT

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.

- 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 *height* attribute.
- MIDDLE: also a *special* adjustment, but the result is the same as for MIDDLE_CENTER.

Text.**get_pos** ()

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, else it's None.

Text.**get_align** ()

Returns the actual text alignment as string, see tables above.

Text.**set_align** (*align*='LEFT')

Just for experts: Sets the text alignment without setting the alignment points, set adjustment points *insert* and *align-point* manually.

Shape

class Shape (*GraphicEntity*)

Shapes (dxftype is SHAPE) are objects that you use like blocks. Shapes are stored in external shape files (*.SHX). You can specify the scale and rotation for each shape reference as you add it. You can not create shapes with *ezdxf*, you can just insert shape references.

Create a *Shape* reference in layouts and blocks by factory function *add_shape* ().

DXF Attributes for Shape

Common DXF Attributes for DXF R12

Common DXF Attributes for DXF R13 or later

Shape.dxf.**insert**

Insertion point as (2D/3D Point in *OCS*)

Shape.dxf.**name**

Shape name

Shape.dxf.**size**

Shape size

Shape.dxf.**rotation**

Rotation angle in degrees; default=0

Shape.dxf.**xscale**

Relative X scale factor; default=1

Shape.dxf.**oblique**

Oblique angle; default=0

Polyline

class Polyline (*GraphicEntity*)

The POLYLINE entity 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 the polyline type by `Polyline.get_mode()`.

Create 2D polylines in layouts and blocks by factory function `add_polyline2D()`.

For 2D entities all points in *OCS*.

Create 3D polylines in layouts and blocks by factory function `add_polyline3D()`.

For 3D entities all points in *WCS*.

DXF Attributes for Polyline

Common DXF Attributes for DXF R12

Common DXF Attributes for DXF R13 or later

`Polyline.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).

`Polyline.dxf.flags`

Constants defined in `ezdxf.const`:

<code>Polyline.dxf.flags</code>	Value	Description
<code>POLYLINE_CLOSED</code>	1	This is a closed Polyline (or a polygon mesh closed in the M direction)
<code>POLYLINE_MESH_CLOSED_M_DIRECTION</code>		equals <code>POLYLINE_CLOSED</code>
<code>POLYLINE_CURVE_FIT_VERTICES_ADDED</code>		Curve-fit vertices have been added
<code>POLYLINE_SPLINE_FIT_VERTICES_ADDED</code>		Spline-fit vertices have been added
<code>POLYLINE_3D_POLYLINE</code>	8	This is a 3D Polyline
<code>POLYLINE_3D_POLYMESH</code>	16	This is a 3D polygon mesh
<code>POLYLINE_MESH_CLOSED_N_DIRECTION</code>		The polygon mesh is closed in the N direction
<code>POLYLINE_POLYFACE_MESH</code>	64	This Polyline is a polyface mesh
<code>POLYLINE_GENERATE_LINETYPE_PATTERN</code>		The linetype pattern is generated continuously around the vertices of this Polyline

`Polyline.dxf.default_start_width`

Default line start width (float); default=0

`Polyline.dxf.default_end_width`

Default line end width (float); default=0

`Polyline.dxf.m_count`

Polymesh M vertex count (int); default=1

`Polyline.dxf.n_count`

Polymesh N vertex count (int); default=1

`Polyline.dxf.m_smooth_density`

Smooth surface M density (int); default=0

`Polyline.dxf.n_smooth_density`

Smooth surface N density (int); default=0

`Polyline.dxf.smooth_type`

Curves and smooth surface type (int); default=0, see table below

Constants for *smooth_type* defined in `ezdxf.const`:

<code>Polyline.dxf.smooth_type</code>	Value	Description
<code>POLYMESH_NO_SMOOTH</code>	0	no smooth surface fitted
<code>POLYMESH_QUADRATIC_BSPLINE</code>	5	quadratic B-spline surface
<code>POLYMESH_CUBIC_BSPLINE</code>	6	cubic B-spline surface
<code>POLYMESH_BEZIER_SURFACE</code>	8	Bezier surface

Polyline Attributes

`Polyline.is_2d_polyline`

True if polyline is a 2D polyline.

`Polyline.is_3d_polyline`

True if polyline is a 3D polyline.

`Polyline.is_polygon_mesh`

True if polyline is a polygon mesh, see *Polymesh*

`Polyline.is_poly_face_mesh`

True if polyline is a poly face mesh, see *Polyface*

`Polyline.is_closed`

True if polyline is closed.

`Polyline.is_m_closed`

True if polyline (as polymesh) is closed in m direction.

`Polyline.is_n_closed`

True if polyline (as polymesh) is closed in n direction.

Polyline Methods

`Polyline.get_mode()`

Returns a string: `AcDb2dPolyline`, `AcDb3dPolyline`, `AcDbPolygonMesh` or `AcDbPolyFaceMesh`

`Polyline.m_close()`

Close mesh in M direction (also closes polylines).

`Polyline.n_close()`

Close mesh in N direction.

`Polyline.close(m_close, n_close=False)`

Close mesh in M (if *mclose* is *True*) and/or N (if *nclose* is *True*) direction.

`Polyline.__len__()`

Returns count of vertices.

`Polyline.__getitem__(pos)`

Get *Vertex* object at position *pos*. Very slow!!!. Vertices are organized as linked list, so it is faster to work with a temporary list of vertices: `list(polyline.vertices())`.

`Polyline.vertices()`

Iterate over all polyline vertices as *Vertex* objects. (replaces `Polyline.__iter__()`)

`Polyline.points()`

Iterate over all polyline points as (x, y[, z])-tuples, not as *Vertex* objects.

`Polyline.append_vertices(points, dxfattribs=None)`

Append points as *Vertex* objects.

param points iterable polyline points, every point is a (x, y[, z])-tuple.

param dxfattribs dict of DXF attributes for the *Vertex*

`Polyline.insert_vertices(pos, points, dxfattribs=None)`

Insert points as *Vertex* objects at position *pos*.

param int pos 0-based insert position

param iterable points iterable polyline points, every point is a tuple.

param dxfattribs dict of DXF attributes for the *Vertex*

`Polyline.delete_vertices(pos, count=1)`

Delete *count* vertices at position *pos*.

param int pos 0-based insert position

param int count count of vertices to delete

Vertex

class Vertex (*GraphicEntity*)

A vertex represents a polyline/mesh point, dxftype is VERTEX, you don't have to create vertices by yourself.

DXF Attributes for Vertex

`Vertex.dxf.location`

vertex location (2D/3D Point *OCS* when 2D, *WCS* when 3D)

`Vertex.dxf.start_width`

line segment start width (float); default=0

`Vertex.dxf.end_width`

line segment end width (float); default=0

`Vertex.dxf.bulge`

Bulge (float); default=0. The bulge is the tangent of one fourth the included angle for an arc segment, made negative if the arc goes clockwise from the start point to the endpoint. A bulge of 0 indicates a straight segment, and a bulge of 1 is a semicircle.

`Vertex.dxf.flags`

Constants defined in `ezdxf.const`:

Vertex.dxf.flags	Value	Description
<code>VTX_EXTRA_VERTEX</code>	<code>CREATED</code>	Extra vertex created by curve-fitting
<code>VTX_CURVE_FIT_TANGENT</code>	<code>TANGENT</code>	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.
<code>VTX_SPLINE_VERTEX</code>	<code>CREATED</code>	Spline vertex created by spline-fitting
<code>VTX_SPLINE_FRAME_CONTROL_POINT</code>	<code>CONTROL_POINT</code>	Spline control point
<code>VTX_3D_POLYLINE_VERTEX</code>	<code>VERTEX</code>	3D polyline vertex
<code>VTX_3D_POLYGON_MESH_VERTEX</code>	<code>VERTEX</code>	Polygon mesh
<code>VTX_3D_POLYFACE_MESH_VERTEX</code>	<code>VERTEX</code>	Face mesh vertex

`Vertex.dxf.tangent`

curve fit tangent direction (float)

`Vertex.dxf.vtx1`

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

`Vertex.dxf.vtx2`

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

`Vertex.dxf.vtx3`

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

`Vertex.dxf.vtx4`

index of 4th vertex, if used as face (feature for experts)

Polymesh

class `Polymesh` (*Polyline*)

A polymesh is a grid of `mcount` x `ncount` vertices and every vertex has its own xyz-coordinates. The *Polymesh* is an extended *Polyline* class, `dxftype` is also `POLYLINE` but `get_mode()` returns `AcDbPolygonMesh`. Create polymeshes in layouts and blocks by factory function `add_polymesh()`.

`Polymesh.get_mesh_vertex(pos)`

Get mesh vertex at position *pos* as *Vertex*.

param pos 0-based (row, col)-tuple

`Polymesh.set_mesh_vertex(pos, point, dxfattribs=None)`

Set mesh vertex at position *pos* to location *point* and update the dxf attributes of the *Vertex*.

param pos 0-based (row, col)-tuple

param point vertex coordinates as (x, y, z)-tuple

param dxfattribs dict of DXF attributes for the *Vertex*

`Polymesh.get_mesh_vertex_cache()`

Get a *MeshVertexCache* object for this Polymesh. The caching object provides fast access to the location attributes of the mesh vertices.

class 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]`

`MeshVertexCache.vertices`

Dict of mesh vertices, keys are 0-based (row, col)-tuples. Writing to this dict doesn't change the DXF entity.

`MeshVertexCache.__getitem__` (*pos*)

Returns the location of *Vertex* at position *pos* as (x, y, z)-tuple

param tuple pos 0-based (row, col)-tuple

`MeshVertexCache.__setitem__` (*pos, location*)

Set the location of *Vertex* at position *pos* to *location*.

param pos 0-based (row, col)-tuple

param location (x, y, z)-tuple

Polyface

class Polyface (*Polyline*)

A polyface consist of multiple location independent 3D areas called faces. The *Polyface* is an extended *Polyline* class, `dxftype` is also POLYLINE but `get_mode()` returns `AcDbPolyFaceMesh`. Create polyfaces in layouts and blocks by factory function `add_polyface()`.

`Polyface.append_face` (*face, dxfattribs=None*)

Append one *face*, *dxfattribs* is used for all vertices generated. Appending single faces is very inefficient, if possible use `append_faces()` to add a list of new faces.

param face a tuple of 3 or 4 3D points, a 3D point is a (x, y, z)-tuple

param dxfattribs dict of DXF attributes for the *Vertex*

`Polyface.append_faces` (*faces, dxfattribs=None*)

Append a list of *faces*, *dxfattribs* is used for all vertices generated.

param tuple faces a list of faces, a face is a tuple of 3 or 4 3D points, a 3D point is a (x, y, z)-tuple

param dxfattribs dict of DXF attributes for the *Vertex*

`Polyface.faces` ()

Iterate over all faces, a face is a tuple of *Vertex* objects; yields (vtx1, vtx2, vtx3[, vtx4], face_record)-tuples

`Polyface.indexed_faces` ()

Returns a list of all vertices and a generator of *Face* () objects as tuple:

```
vertices, faces = polyface.indexed_faces()
```

`Polyface.optimize` (*precision=6*)

Rebuilds *Polyface* with vertex optimization. Merges vertices with nearly same vertex locations. Polyfaces created by *ezdxf* are optimized automatically.

param int precision decimal precision for determining identical vertex locations

See also:

Tutorial for Polyface

class Face

Represents a single face of the *Polyface* entity.

Face.**vertices**

List of all *Polyface* vertices (without *face_records*). (read only attribute)

Face.**face_record**

The face forming vertex of type *AcDbFaceRecord*, contains the indices to the face building vertices. Indices of the DXF structure are 1-based and a negative index indicates the beginning of an invisible edge. *Face.face_record.dxf.color* determines the color of the face. (read only attribute)

Face.**indices**

Indices to the face forming vertices as tuple. This indices are 0-base and are used to get vertices from the list *Face.vertices*. (read only attribute)

Face.**__iter__**()

Iterate over all face vertices as *Vertex* objects.

Face.**__len__**()

Returns count of face vertices (without *face_record*).

Face.**__getitem__**(*pos*)

Returns *Vertex* at position *pos*.

param int pos vertex position 0-based

Face.**points**()

Iterate over all face vertex locations as (x, y, z)-tuples.

Face.**is_edge_visible**(*pos*)

Returns *True* if edge starting at vertex *pos* is visible else *False*.

param int pos vertex position 0-based

Solid

class Solid(*GraphicEntity*)

A solid filled triangle or quadrilateral, *dxftype* is *SOLID*. Access corner points by name (*entity.dxf.vtx0 = (1.7, 2.3)*) or by index (*entity[0] = (1.7, 2.3)*). Create solids in layouts and blocks by factory function *add_solid()*.

DXF Attributes for Solid

Common DXF Attributes for DXF R12

Common DXF Attributes for DXF R13 or later

`Solid.dxf.vtx0`

location of the 1. point (2D/3D Point in *OCS*)

`Solid.dxf.vtx1`

location of the 2. point (2D/3D Point in *OCS*)

`Solid.dxf.vtx2`

location of the 3. point (2D/3D Point in *OCS*)

`Solid.dxf.vtx3`

location of the 4. point (2D/3D Point in *OCS*)

Trace

class Trace (*GraphicEntity*)

A Trace is solid filled triangle or quadrilateral, dxftype is TRACE. Access corner points 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. Create traces in layouts and blocks by factory function `add_trace()`.

DXF Attributes for Trace

Common DXF Attributes for DXF R12

Common DXF Attributes for DXF R13 or later

`Trace.dxf.vtx0`

location of the 1. point (2D/3D Point in *OCS*)

`Trace.dxf.vtx1`

location of the 2. point (2D/3D Point in *OCS*)

`Trace.dxf.vtx2`

location of the 3. point (2D/3D Point in *OCS*)

`Trace.dxf.vtx3`

location of the 4. point (2D/3D Point in *OCS*)

3DFace

class 3DFace (*GraphicEntity*)

(This is not a valid Python name, but it works, because all classes described here, do not exist in this simple form.)

A 3DFace is real 3D solid filled triangle or quadrilateral, dxftype is 3DFACE. Access corner points by name (`entity.dxf.vtx0 = (1.7, 2.3)`) or by index (`entity[0] = (1.7, 2.3)`). Create 3DFaces in layouts and blocks by factory function `add_3dface()`.

DXF Attributes for 3DFace

Common DXF Attributes for DXF R12

Common DXF Attributes for DXF R13 or later

3DFace.dxf.**vtx0**

location of the 1. point (3D Point in *WCS*)

3DFace.dxf.**vtx1**

location of the 2. point (3D Point in *WCS*)

3DFace.dxf.**vtx2**

location of the 3. point (3D Point in *WCS*)

3DFace.dxf.**vtx3**

location of the 4. point (3D Point in *WCS*)

3DFace.dxf.**invisible_edge**

invisible edge flag (int, default=0)

- 1 = first edge is invisible
- 2 = second edge is invisible
- 4 = third edge is invisible
- 8 = fourth edge is invisible

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

Ellipse

class Ellipse (*GraphicEntity*)

Introduced in AutoCAD R13 (DXF version AC1012), *dxftype* is ELLIPSE.

An ellipse with center point at location *center* and a major axis *major_axis* as vector. *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 2π . The ellipse goes from start to end param in *counter clockwise* direction. Create ellipses in layouts and blocks by factory function `add_ellipse()`.

`Ellipse.dxf.extrusion` is supported, but does not establish an *OCS*, it is used to create an 3D entity by extruding the base ellipse.

DXF Attributes for Ellipse

Common DXF Attributes for DXF R13 or later

Ellipse.dxf.**center**

center point of circle (2D/3D Point in *WCS*)

Ellipse.dxf.**major_axis**

Endpoint of major axis, relative to the center (tuple of float)

Ellipse.dxf.**ratio**

Ratio of minor axis to major axis (float)

`Ellipse.dxf.start_param`

Start parameter (this value is 0.0 for a full ellipse) (float)

`Ellipse.dxf.end_param`

End parameter (this value is 2π for a full ellipse) (float)

LWPolyline

class `LWPolyline` (*GraphicEntity*)

Introduced in DXF version R13 (AC1012), dxftype is LWPOLYLINE.

A lightweight polyline is defined as a single graphic entity. The *LWPolyline* differs from the old-style *Polyline*, which is defined as a group of subentities. *LWPolyline* display faster (in AutoCAD) and consume less disk space and RAM. Create *LWPolyline* in layouts and blocks by factory function `add_lwpolyline()`. *LWPolyline* is a planar element, therefore all points in *OCS* as (x, y) tuples (*elevation* is the z-axis value).

Bulge Value

The bulge value is used to create arc shaped line segments. The bulge 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 (count clockwise)
- negative value (< 0): bulge is left of line (clockwise)
- 0 = no bulge

Start Width 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 vertex has only a meaning if the polyline is closed, and they apply to the last line segment from the last vertex to the first vertex.

See also:

Tutorial for LWPolyline

User Defined Point Format Codes

Code	Point Component
x	x coordinate
y	y coordinate
s	start width
e	end width
b	bulge value
v	(x, y) as tuple

DXF Attributes for LWPolyline

Common DXF Attributes for DXF R13 or later

`LWPolyline.dxf.elevation`

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

`LWPolyline.dxf.flags`

Constants defined in `ezdxf.const`:

<code>LWPolyline.dxf.flags</code>	Value	Description
<code>LWPOLYLINE_CLOSED</code>	1	polyline is closed
<code>LWPOLYLINE_PLINEGEN</code>	128	???

`LWPolyline.dxf.const_width`

constant line width (float), default=0

`LWPolyline.dxf.count`

number of vertices (read only), same as `len(polyline)`

LWPolyline Attributes

`LWPolyline.closed`

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

LWPolyline Methods

`LWPolyline.get_points` (*format*='xyseb')

Parameters `format` – format string, see *format codes*

Returns all polyline points as list of tuples (x, y, start_width, end_width, bulge), format specifies a user defined point format.

start_width, end_width and bulge is 0 if not present (0 is the DXF default value if not present).

All points in *OCS* as (x, y) tuples (elevation is the z-axis value).

`LWPolyline.set_points` (*points*, *format*='xyseb')

Parameters `format` – format string, see *format codes*

Replace existing polyline points by new *points*, *points* is a list of (x, y, [start_width, [end_width, [bulge]]]) tuples. Set start_width, end_width to 0 to be ignored (x, y, 0, 0, bulge).

All points in *OCS* as (x, y) tuples (elevation is the z-axis value).

LWPPolyline.**points** (*format*='xyseb')

Parameters *format* – format string, see *format codes*

Context manager for polyline points. Returns a standard Python list of points, according to the format string.

All coordinates in *OCS*.

LWPPolyline.**vertices** ()

Yield all polyline points as (x, y) tuples in *OCS* (elevation is the z-axis value).

LWPPolyline.**vertices_in_wcs** ()

Yield all polyline points as (x, y, z) tuples in *WCS*.

LWPPolyline.**append** (*point*, *format*='xyseb')

Parameters *format* – format string, see *format codes*

Append new point, format specifies a user defined point format.

All coordinates in *OCS*.

LWPPolyline.**append_points** (*points*, *format*='xyseb')

Parameters

- **points** – iterable of point, point is (x, y, [start_width, [end_width, [bulge]]]) tuple
- **format** – format string, see *format codes*

Append new points, points is a list of (x, y, [start_width, [end_width, [bulge]]]) tuples. Set start_width, end_width to 0 to be ignored (x, y, 0, 0, bulge).

All coordinates in *OCS*.

LWPPolyline.**insert** (*pos*, *point*, *format*='xyseb')

Parameters

- **pos** – insertion position for new point
- **point** – new polyline point
- **format** – format string, see *format codes*

Insert new point in front of position *pos*, format specifies a user defined point format.

All coordinates in *OCS*.

LWPPolyline.**clear** ()

Remove all points.

LWPPolyline.**__len__** ()

Number of polyline points.

LWPPolyline.**__getitem__** (*index*)

Get point at position *index* as (x, y, start_width, end_width, bulge) tuple. start_width, end_width and bulge is 0 if not present (0 is the DXF default value if not present), supports extended slicing. Point format is fixed as 'xyseb'.

All coordinates in *OCS*.

LWPolyline. **__setitem__** (*index*, *value*)

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*.

LWPolyline. **__delitem__** (*index*)

Delete point at position *index*, supports extended slicing.

MText

class MText (*GraphicEntity*)

Introduced in DXF version R13 (AC1012), extended in DXF version R2007 (AC1021), dxftype is MTEXT.

Multiline text fits a specified width but can extend vertically to an indefinite length. You can format individual words or characters within the MText. Create *MText* in layouts and blocks by factory function *add_mtext()*.

See also:

Tutorial for MText

DXF Attributes for MText

Common DXF Attributes for DXF R13 or later

MText.dxf.**insert**

Insertion point (3D Point in *OCS*)

MText.dxf.**char_height**

Initial text height (float); default=1.0

MText.dxf.**width**

Reference rectangle width (float)

MText.dxf.**attachment_point**

Constants defined in *ezdxf.const*:

MText.dxf.attachment_point	Value
MTEXT_TOP_LEFT	1
MTEXT_TOP_CENTER	2
MTEXT_TOP_RIGHT	3
MTEXT_MIDDLE_LEFT	4
MTEXT_MIDDLE_CENTER	5
MTEXT_MIDDLE_RIGHT	6
MTEXT_BOTTOM_LEFT	7
MTEXT_BOTTOM_CENTER	8
MTEXT_BOTTOM_RIGHT	9

MText.dxf.**flow_direction**

Constants defined in *ezdxf.const*:

MText.dxf.flow_direction	Value	Description
MTEXT_LEFT_TO_RIGHT	1	left to right
MTEXT_TOP_TO_BOTTOM	3	top to bottom
MTEXT_BY_STYLE	5	by style (the flow direction is inherited from the associated text style)

MText.dxf.style

Text style (string); default='STANDARD'

MText.dxf.text_direction

X-axis direction vector in *WCS* (3D Point); default=(1, 0, 0); if rotation and text_direction are present, text_direction wins

MText.dxf.rotation

Text rotation in degrees (float); default=0

MText.dxf.line_spacing_style

line spacing style (int), see table below

MText.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.const`:

MText.dxf.line_spacing_style	Value	Description
MTEXT_AT_LEAST	1	taller characters will override
MTEXT_EXACT	2	taller characters will not override

MText Methods

MText.get_text()

Returns content of *MText* as string.

MText.set_text(text)

Set *text* as *MText* content.

MText.set_location(insert, rotation=None, attachment_point=None)

Set DXF attributes *insert*, *rotation* and *attachment_point*, *None* for *rotation* or *attachment_point* preserves the existing value.

MText.get_rotation()

Get text rotation in degrees, independent if it is defined by *rotation* or *text_direction*

MText.set_rotation(angle)

Set DXF attribute *rotation* to *angle* (in degrees) and deletes *text_direction* if present.

MText.edit_data()

Context manager for *MText* content:

```

with mtext.edit_data() as data:
    data += "append some text" + data.NEW_LINE

    # or replace whole text
    data.text = "Replacement for the existing text."

```

MTextData

class MTextData

Temporary object to manage the *MText* content. Create context object by *MText.edit_data()*.

See also:

Tutorial for MText

MTextData.text

Represents the *MText* content, treat it like a normal string. (read/write)

MTextData.__iadd__(text)

Append *text* to the *MTextData.text* attribute.

MTextData.append(text)

Synonym for *MTextData.__iadd__()*.

MTextData.set_font(name, bold=False, italic=False, codepage=1252, pitch=0)

Change actual font inline.

MTextData.set_color(color_name)

Set text color to red, yellow, green, cyan, blue, magenta or white.

Convenient constants defined in MTextData:

Constant	Description
UNDERLINE_START	start underline text (b += b.UNDERLINE_START)
UNDERLINE_STOP	stop underline text (b += b.UNDERLINE_STOP)
UNDERLINE	underline text (b += b.UNDERLINE % "Text")
OVERSTRIKE_START	start overstrike
OVERSTRIKE_STOP	stop overstrike
OVERSTRIKE	overstrike text
STRIKE_START	start strike trough
STRIKE_STOP	stop strike trough
STRIKE	strike trough text
GROUP_START	start of group
GROUP_END	end of group
GROUP	group text
NEW_LINE	start in new line (b += "Text" + b.NEW_LINE)
NBSP	none breaking space (b += "Python" + b.NBSP + "3.4")

Hatch

class Hatch

Introduced in DXF version R13 (AC1012), *dxf* type is HATCH.

Fills an enclosed area defined by one or more boundary paths with a hatch pattern, solid fill, or gradient fill. Create *Hatch* in layouts and blocks by factory function *add_hatch()*.

All points in *OCS* as (x, y) tuples (*elevation* is the z-axis value).

DXF Attributes for Hatch

Common DXF Attributes for DXF R13 or later

`Hatch.dxf.pattern_name`
 pattern name as string

`Hatch.dxf.solid_fill`

- 1 = solid fill, better use: *Hatch.set_solid_fill()*
- 0 = pattern fill, better use: *Hatch.set_pattern_fill()*

`Hatch.dxf.associative`

- 1 = associative hatch
- 0 = not associative hatch

Associations not handled by ezdxf, you have to set the handles to the associated DXF entities by yourself.

`Hatch.dxf.hatch_style`

- 0 = normal
- 1 = outer
- 2 = ignore

(search for AutoCAD help for more information)

`Hatch.dxf.pattern_type`

- 0 = user
- 1 = predefined
- 2 = custom (???)

`Hatch.dxf.pattern_angle`

Pattern angle in degrees (360 deg = circle)

`Hatch.dxf.pattern_scale`

`Hatch.dxf.pattern_double`

1 = double else 0

`Hatch.dxf.n_seed_points`

Count of seed points (better user: *Hatch.get_seed_points()*)

`Hatch.dxf.elevation`

Z value represents the elevation height of the *OCS*

Hatch Attributes

`Hatch.has_solid_fill`

True if hatch has a solid fill else *False*. (read only)

`Hatch.has_pattern_fill`

True if hatch has a pattern fill else *False*. (read only)

`Hatch.has_gradient_fill`

True if hatch has a gradient fill else *False*. A hatch with gradient fill has also a solid fill. (read only)

`Hatch.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
hatch.bgcolor = (10, 20, 30) # set background color
del hatch.bgcolor # delete background color
```

`Hatch.edit_boundary()`

Context manager to edit hatch boundary data, yields a *BoundaryPathData* object.

`Hatch.edit_pattern()`

Context manager to edit hatch pattern data, yields a *PatternData* object.

`Hatch.set_pattern_definition(lines)`

Setup hatch pattern definition by a list of definition lines and a definition line is a 4-tuple [angle, base_point, offset, dash_length_items]

- *angle*: line angle in degrees
- *base-point*: (x, y) tuple
- *offset*: (dx, dy) tuple, added to base point for next line and so on
- *dash_length_items*: list of dash items (item > 0 is a line, item < 0 is a gap and item == 0.0 is a point)

Parameters *lines* (*list*) – list of definition lines

`Hatch.set_solid_fill(color=7, style=1, rgb=None)`

Set *Hatch* to solid fill mode and removes all gradient and pattern fill related data.

Parameters

- **color** (*int*) – ACI (AutoCAD Color Index) in range 0 to 256, (0 = BYBLOCK; 256 = BYLAYER)
- **style** (*int*) – hatch style (0 = normal; 1 = outer; 2 = ignore)
- **rgb** (*tuple*) – true color value as (r, g, b) tuple - has higher priority than *color*. True color support requires at least DXF version AC1015.

`Hatch.set_gradient(color1=(0, 0, 0), color2=(255, 255, 255), rotation=0., centered=0., one_color=0, tint=0., name='LINEAR')`

Set *Hatch* to gradient fill mode and removes all pattern fill related data. Gradient support requires at least DXF version AC1018. A gradient filled hatch is also a solid filled hatch.

Parameters

- **color1** (*tuple*) – (r, g, b) tuple for first color, rgb values as int in range 0..255
- **color2** (*tuple*) – (r, g, b) tuple for second color, rgb values as int in range 0..255

- **rotation** (*float*) – rotation in degrees (360 deg = circle)
- **centered** (*int*) – determines whether the gradient is centered or not
- **one_color** (*int*) – 1 for gradient from *color1* to tinted *color1*
- **tint** (*float*) – determines the tinted target *color1* for a one color gradient. (valid range 0.0 to 1.0)
- **name** (*str*) – name of gradient type, default ‘LINEAR’

Valid gradient type names are:

- LINEAR
- CYLINDER
- INVCYLINDER
- SPHERICAL
- INVSPHERICAL
- HEMISPHERICAL
- INVHEMISPHERICAL
- CURVED
- INVCURVED

`Hatch.get_gradient()`
Get gradient data, returns a *GradientData* object.

`Hatch.edit_gradient()`
Context manager to edit hatch gradient data, yields a *GradientData* object.

`Hatch.set_pattern_fill(name, color=7, angle=0., scale=1., double=0, style=1, pattern_type=1, definition=None)`
Set *Hatch* to pattern fill mode. Removes all gradient related data.

Parameters

- **color** (*int*) – AutoCAD Color Index in range 0 to 256, (0 = BYBLOCK; 256 = BY-LAYER)
- **angle** (*float*) – angle of pattern fill in degrees (360 deg = circle)
- **scale** (*float*) – pattern scaling
- **double** (*int*) – double flag
- **style** (*int*) – hatch style (0 = normal; 1 = outer; 2 = ignore)
- **pattern_type** (*int*) – pattern type (0 = user-defined; 1 = predefined; 2 = custom) ???
- **definition** (*list*) – list of definition lines and a definition line is a 4-tuple [angle, base_point, offset, dash_length_items], see *Hatch.set_pattern_definition()*

`Hatch.get_seed_points()`
Get seed points as list of (x, y) points, I don’t know why there can be more than one seed point. All points in *OCS* (*elevation* is the Z value).

`Hatch.set_seed_points(points)`
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* (*elevation* is the Z value)

See also:

Tutorial for Hatch

Hatch Boundary Helper Classes

class BoundaryPathData

Defines the borders of the hatch, a hatch can consist of more than one path.

BoundaryPathData.paths

List of all boundary paths. Contains *PolylinePath* and *EdgePath* objects. (read/write)

BoundaryPathData.add_polyline_path(path_vertices, is_closed=1, flags=1)

Create and add a new *PolylinePath* object.

Parameters

- **path_vertices** (*list*) – list of polyline vertices as (x, y) or (x, y, bulge) tuples.
- **is_closed** (*int*) – 1 for a closed polyline else 0
- **flags** (*int*) – external(1) or outermost(16) or default (0)

BoundaryPathData.add_edge_path(flags=1)

Create and add a new *EdgePath* object.

Parameters flags (*int*) – external(1) or outermost(16) or default (0)

BoundaryPathData.clear()

Remove all boundary paths.

class PolylinePath

A polyline as hatch boundary path.

PolylinePath.path_type_flags

(bit coded flags)

- 0 = default
- 1 = external
- 2 = polyline, will be set by *ezdxf*
- 16 = outermost

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 contains the pixel size tag (47) - delete it `hatch.AcDbHatch.remove_tags([47])` and maybe the problem is solved. *ezdxf* does not use the pixel size tag, but it can occur in DXF files created by other applications.

PolylinePath.is_closed

True if polyline path is closed else *False*.

PolylinePath.vertices

List of path vertices as (x, y, bulge) tuples. (read/write)

PolylinePath.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)

PolylinePath.set_vertices (*vertices, is_closed=1*)

Set new vertices for the polyline path, a vertex has to be a (x, y) or a (x, y, bulge) tuple.

PolylinePath.clear ()

Removes all vertices and all links to associated DXF objects (*PolylinePath.source_boundary_objects*).

class 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.

EdgePath.path_type_flags

(bit coded flags)

- 0 = default
- 1 = external
- 16 = outermost

see *PolylinePath.path_type_flags*

EdgePath.edges

List of boundary edges of type *LineEdge*, *ArcEdge*, *EllipseEdge* of *SplineEdge*

EdgePath.source_boundary_objects

Required for associative hatches, list of handles to the associated DXF entities.

EdgePath.clear ()

Delete all edges.

EdgePath.add_line (*start, end*)

Add a *LineEdge* from *start* to *end*.

Parameters

- **start** (*tuple*) – start point of line, (x, y) tuple
- **end** (*tuple*) – end point of line, (x, y) tuple

EdgePath.add_arc (*center, radius=1., start_angle=0., end_angle=360., is_counter_clockwise=0*)

Add an *ArcEdge*.

Parameters

- **center** (*tuple*) – center point of arc, (x, y) tuple
- **radius** (*float*) – radius of circle
- **start_angle** (*float*) – start angle of arc in degrees
- **end_angle** (*float*) – end angle of arc in degrees
- **is_counter_clockwise** (*int*) – 1 for yes 0 for no

EdgePath.add_ellipse (*center, major_axis_vector=(1., 0.), minor_axis_length=1., start_angle=0., end_angle=360., is_counter_clockwise=0*)

Add an *EllipseEdge*.

Parameters

- **center** (*tuple*) – center point of ellipse, (x, y) tuple

- **major_axis** (*tuple*) – vector of major axis as (x, y) tuple
- **ratio** (*float*) – ratio of minor axis to major axis as float
- **start_angle** (*float*) – start angle of ellipse in degrees
- **end_angle** (*float*) – end angle of ellipse in degrees
- **is_counter_clockwise** (*int*) – 1 for yes 0 for no

EdgePath.**add_spline** (*fit_points=None, control_points=None, knot_values=None, weights=None, degree=3, rational=0, periodic=0*)

Add a *SplineEdge*.

Parameters

- **fit_points** (*list*) – points through which the spline must go, at least 3 fit points are required. list of (x, y) tuples
- **control_points** (*list*) – affects the shape of the spline, mandatory and AutoCAD crashes on invalid data. list of (x, y) tuples
- **knot_values** (*list*) – (knot vector) mandatory and AutoCAD crashes on invalid data. list of floats; *ezdxf* provides two tool functions to calculate valid knot values: `ezdxf.tools.knot_values(n_control_points, degree)` and `ezdxf.tools.knot_values_uniform(n_control_points, degree)`
- **weights** (*list*) – weight of control point, not mandatory, list of floats.
- **degree** (*int*) – degree of spline
- **rational** (*int*) – 1 for rational spline, 0 for none rational spline
- **periodic** (*int*) – 1 for periodic spline, 0 for none periodic spline

Warning: Unlike for the spline entity AutoCAD does not calculate the necessary *knot_values* for the spline edge itself. On the contrary, if the *knot_values* in the spline edge are missing or invalid AutoCAD **crashes**.

class LineEdge

Straight boundary edge.

LineEdge.start

Start point as (x, y) tuple. (read/write)

LineEdge.end

End point as (x, y) tuple. (read/write)

class ArcEdge

Arc as boundary edge.

ArcEdge.center

Center point of arc as (x, y) tuple. (read/write)

ArcEdge.radius

Arc radius as float. (read/write)

ArcEdge.start_angle

Arc start angle in degrees (360 deg = circle). (read/write)

ArcEdge.end_angle

Arc end angle in degrees (360 deg = circle). (read/write)

`ArcEdge.is_counter_clockwise`

1 for counter clockwise arc else 0. (read/write)

class EllipseEdge

Elliptic arc as boundary edge.

`EllipseEdge.major_axis_vector`

Ellipse major axis vector as (x, y) tuple. (read/write)

`EllipseEdge.minor_axis_length`

Ellipse minor axis length as float. (read/write)

`EllipseEdge.radius`

Ellipse radius as float. (read/write)

`EllipseEdge.start_angle`

Ellipse start angle in degrees (360 deg = circle). (read/write)

`EllipseEdge.end_angle`

Ellipse end angle in degrees (360 deg = circle). (read/write)

`EllipseEdge.is_counter_clockwise`

1 for counter clockwise ellipse else 0. (read/write)

class SplineEdge

Spline as boundary edge.

`SplineEdge.degree`

Spline degree as int. (read/write)

`SplineEdge.rational`

1 for rational spline else 0. (read/write)

`SplineEdge.periodic`

1 for periodic spline else 0. (read/write)

`SplineEdge.knot_values`

List of knot values as floats. (read/write)

`SplineEdge.control_points`

List of control points as (x, y) tuples. (read/write)

`SplineEdge.fit_points`

List of fit points as (x, y) tuples. (read/write)

`SplineEdge.weights`

List of weights (of control points) as floats. (read/write)

`SplineEdge.start_tangent`

Spline start tangent (vector) as (x, y) tuple. (read/write)

`SplineEdge.end_tangent`

Spline end tangent (vector) as (x, y) tuple. (read/write)

Hatch Pattern Definition Helper Classes

class PatternData

`PatternData.lines`

List of pattern definition lines (read/write). see *PatternDefinitionLine*

`PatternData.new_line` (*angle=0., base_point=(0., 0.), offset=(0., 0.), dash_length_items=None*)
 Create a new pattern definition line, but does not add the line to the `PatternData.lines` attribute.

`PatternData.add_line` (*angle=0., base_point=(0., 0.), offset=(0., 0.), dash_length_items=None*)
 Create a new pattern definition line and add the line to the `PatternData.lines` attribute.

`PatternData.clear` ()
 Delete all pattern definition lines.

class PatternDefinitionLine

Represents a pattern definition line, use factory function `PatternData.new_line()` to create new pattern definition lines.

`PatternDefinitionLine.angle`
 Line angle in degrees (circle = 360 deg). (read/write)

`PatternDefinitionLine.base_point`
 Base point as (x, y) tuple. (read/write)

`PatternDefinitionLine..offset`
 Offset as (x, y) tuple. (read/write)

`PatternDefinitionLine.dash_length_items`
 List of dash length items (item > 0 is line, < 0 is gap, 0.0 = dot). (read/write)

Hatch Gradient Fill Helper Classes

class GradientData

`GradientData.color1`
 First rgb color as (r, g, b) tuple, rgb values in range 0 to 255. (read/write)

`GradientData.color2`
 Second rgb color as (r, g, b) tuple, rgb values in range 0 to 255. (read/write)

`GradientData.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)

`GradientData.rotation`
 Gradient rotation in degrees (circle = 360 deg). (read/write)

`GradientData.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)

`GradientData.tint`
 Specifies the tint (color1 mixed with white) of a color to be used for a gradient fill of one color. (read/write)

See also:

[Tutorial for Hatch Pattern Definition](#)

Mesh

class Mesh (*GraphicEntity*)

Introduced in DXF version R13 (AC1012), dxftype is MESH.

3D mesh entity similar to the `Polyface` entity. Create `Mesh` in layouts and blocks by factory function `add_mesh()`.

All points in *WCS* as (x, y, z) tuples

DXF Attributes for Mesh

Common DXF Attributes for DXF R13 or later

`Mesh.dxf.version`

`Mesh.dxf.blend_crease`

0 = off, 1 = on

`Mesh.dxf.subdivision_levels`

int >= 0, 0 = no smoothing

Mesh Methods

`Mesh.edit_data()`

Context manager various mesh data, returns *MeshData*.

See also:

Tutorial for Image and ImageDef

MeshData

class MeshData

`MeshData.vertices`

A standard Python list with (x, y, z) tuples (read/write)

`MeshData.faces`

A standard Python list with (v1, v2, v3, ...) tuples (read/write)

Each face consist of a list of vertex indices (= index in *MeshData.vertices*).

`MeshData.edges`

A standard Python list with (v1, v2) tuples (read/write)

Each edge consist of exact two vertex indices (= index in *MeshData.vertices*).

`MeshData.edge_crease_values`

A standard Python list of float values, one value for each edge. (read/write)

`MeshData.add_face(vertices)`

Add a face by coordinates, vertices is a list of (x, y, z) tuples.

`MeshData.add_edge(vertices)`

Add an edge by coordinates, vertices is a list of two (x, y, z) tuples.

`MeshData.optimize(precision=6)`

Tries to reduce vertex count by merging near vertices. *precision* defines the decimal places for coordinate be equal to merge two vertices.

See also:

Tutorial for Mesh

Spline

class Spline (*GraphicEntity*)

Introduced in DXF version R13 (AC1012), dxftype is SPLINE.

A spline curve, all coordinates have to be 3D coordinates even the spline is only a 2D planar curve.

The spline curve is defined by a set of *fit points*, the spline curve passes all these fit points. The *control points* defines a polygon which influences the form of the curve, the first control point should be identical with the first fit point and the last control point should be identical the last fit point.

Don't ask me about the meaning of *knot values* or *weights* and how they influence the spline curve, I don't know it, ask your math teacher or the internet. I think the *knot values* can be ignored, they will be calculated by the CAD program that processes the DXF file and the weights determines the influence 'strength' of the *control points*, in normal case the weights are all 1 and can be left off.

To create a *Spline* curve you just need a bunch of *fit points*, *control point*, *knot_values* and *weights* are optional (tested with AutoCAD 2010). If you add additional data, be sure that you know what you do.

Create *Spline* in layouts and blocks by factory function `add_spline()`.

For more information about spline mathematics go to [Wikipedia](#).

DXF Attributes for Spline

All points in *WCS* as (x, y, z) tuples

Common DXF Attributes for DXF R13 or later

`Spline.dxf.degree`

Degree of the spline curve (int)

`Spline.dxf.flags`

Bit coded option flags, constants defined in `ezdxf.const`:

Spline.dxf.flags	Value	Description
CLOSED_SPLINE	1	Spline is closed
PERIODIC_SPLINE	2	
RATIONAL_SPLINE	4	
PLANAR_SPLINE	8	
LINEAR_SPLINE	16	planar bit is also set

`Spline.dxf.n_knots`

Count of knot values (int), automatically set by `ezdxf` (read only)

`Spline.dxf.n_fit_points`

Count of fit points (int), automatically set by `ezdxf` (read only)

`Spline.dxf.n_control_points`

Count of control points (int), automatically set by `ezdxf` (read only)

Spline.dxf.knot_tolerance
Knot tolerance (float); default=1e-10

Spline.dxf.fit_tolerance
Fit tolerance (float); default=1e-10

Spline.dxf.control_point_tolerance
Control point tolerance (float); default=1e-10

Spline.dxf.start_tangent
Start tangent vector as (3D Point in *WCS*)

Spline.dxf.end_tangent
End tangent vector as (3D Point in *WCS*)

See also:

Tutorial for Spline

Spline Attributes

Spline.closed
True if spline is closed else False. A closed spline has a connection from the last control point to the first control point. (read/write)

Spline.control_points
Returns the control points as *ControlPoints* object in *WCS*.

Spline.fit_points
Returns the fit points as *FitPoints* object in *WCS*.

Spline.knot_values
Returns the knot values as `array.array('f')`.

Spline.weights
Returns the control point weights as `array.array('f')`.

Spline Methods

Spline.set_control_points (*points*)
Set control points, *points* is a list (container or generator) of (x, y, z) tuples in *WCS*.

Spline.set_fit_points (*points*)
Set fit points, *points* is a list (container or generator) of (x, y, z) tuples in *WCS*.

Spline.set_knot_values (*values*)
Set knot values, *values* is a list (container or generator) of *floats*.

Spline.set_weights (*values*)
Set weights, *values* is a list (container or generator) of *floats*.

Spline.set_open_uniform (*control_points*, *degree=3*)
Open B-spline with uniform knot vector, start and end at your first and last control points.

Spline.set_uniform (*control_points*, *degree=3*)
B-spline with uniform knot vector, does NOT start and end at your first and last control points.

Spline.set_periodic (*control_points*, *degree=3*)
Closed B-spline with uniform knot vector, start and end at your first control point.

`Spline.set_open_rational` (*control_points*, *weights*, *degree=3*)

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.

`Spline.set_uniform_rational` (*control_points*, *weights*, *degree=3*)

Rational B-spline with uniform knot vector, does NOT start and end at your first and last control points, and has additional control possibilities by weighting each control point.

`Spline.set_periodic_rational` (*control_points*, *weights*, *degree=3*)

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.

`Spline.edit_data` ()

Context manager for all spline data, returns *SplineData*.

Fit points, control points, knot values and weights can be manipulated as lists by using the general context manager `Spline.edit_data()`:

```
with spline.edit_data() as spline_data:
    # spline_data contains list like objects: add, change or delete items as you
    ↪ want
    # fit_points and control_points have to be (x, y, z) tuples
    # knot_values and weights have to be numbers
    spline_data.fit_points.append((200, 300, 0)) # append a fit point
    # on exit the context manager sets spline data automatically and updates all
    ↪ counters
```

SplineData

class SplineData

`SplineData.fit_points`

FitPoints object with list like behavior.

`SplineData.control_points`

ControlPoints object with list like behavior.

`SplineData.knot_values`

Spline knot values as `array.array('f')`.

`SplineData.weights`

Spline weights as `array.array('f')`.

ControlPoints

A list like object to store vertices as `array.array('d')` flat list.

Supports most standard list operations like indexing, iteration, insert, append, extend and so on.

class ControlPoints (*VertexArray*)

For attributes and methods see *VertexArray*

FitPoints

class FitPoints (*VertexArray*)

Same as *ControlPoints*.

For attributes and methods see *VertexArray*

Image

class Image (*GraphicEntity*)

Introduced in DXF version R13 (AC1012), dxfname is IMAGE.

Add a raster image 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*. Create *Image* in layouts and blocks by factory function *add_image()*. ezdxf creates only images in the XY-plan. You can place images in the 3D space too, but then you have to set the *u_pixel* and the *v_pixel* vectors by yourself.

DXF Attributes for Image

Common DXF Attributes for DXF R13 or later

`Image.dxf.insert`

Insertion point, lower left corner of the image (3D Point in *WCS*).

`Image.dxf.u_pixel`

U-vector of a single pixel (points along the visual bottom of the image, starting at the insertion point) (x, y, z) tuple

`Image.dxf.v_pixel`

V-vector of a single pixel (points along the visual left side of the image, starting at the insertion point) (x, y, z) tuple

`Image.dxf.image_size`

Image size in pixels

`Image.dxf.image_def`

Handle to the image definition entity, see *ImageDef*

`Image.dxf.flags`

Image.dxf.flags	Value	Description
Image.SHOW_IMAGE	1	Show image
Image.SHOW_WHEN_NOT_ALIGNED	2	Show image when not aligned with screen
Image.USE_CLIPPING_BOUNDARY	4	Use clipping boundary
Image.USE_TRANSPARENCY	8	Transparency is on

`Image.dxf.clipping`

Clipping state: 0 = Off; 1 = On

`Image.dxf.brightness`

Brightness value (0-100; default = 50)

`Image.dxf.contrast`

Contrast value (0-100; default = 50)

`Image.dxf.fade`

Fade value (0-100; default = 0)

`Image.dxf.clipping_boundary_type`

Clipping boundary type. 1 = Rectangular; 2 = Polygonal

`Image.dxf.count_boundary_points`

Number of clip boundary vertices

`Image.dxf.clip_mode`

Clip mode: 0 = Outside; 1 = Inside (R2000)

Image Methods

`Image.get_boundary ()`

Returns a list of vertices as pixel coordinates, lower left corner is (0, 0) and upper right corner is (ImageSizeX, ImageSizeY), independent from the absolute location of the image in WCS.

`Image.reset_boundary ()`

Reset boundary path to the default rectangle [(0, 0), (ImageSizeX, ImageSizeY)].

`Image.set_boundary (vertices)`

Set boundary path to vertices. 2 points describe a rectangle (lower left and upper right corner), more than 2 points is a polygon as clipping path. Sets clipping state to 1 and also sets the Image.USE_CLIPPING_BOUNDARY flag.

`Image.get_image_def ()`

returns the associated IMAGEDEF entity. see *ImageDef*.

Ray

class Ray (*GraphicEntity*)

Introduced in DXF version R13 (AC1012), dxftype is RAY.

A *Ray* starts at a point and continues to infinity. Create *Ray* in layouts and blocks by factory function *add_ray()*.

DXF Attributes for Ray

Common DXF Attributes for DXF R13 or later

`Ray.dxf.start`

Start point as (3D Point in WCS)

`Ray.dxf.unit_vector`

Unit direction vector as (3D Point in WCS)

XLine

class XLine (*GraphicEntity*)

Introduced in DXF version R13 (AC1012), dxftype is XLINE.

A line that extents to infinity in both directions, used as construction line. Create *XLine* in layouts and blocks by factory function *add_xline()*.

DXF Attributes for XLine

Common DXF Attributes for DXF R13 or later

`XLine.dxf.start`

Location point of line as (3D Point in *WCS*)

`XLine.dxf.unit_vector`

Unit direction vector as (3D Point in *WCS*)

Body

class `Body` (*GraphicEntity*)

Introduced in DXF version R13 (AC1012), dxftype is BODY.

A 3D object created by an ACIS based geometry kernel provided by the [Spatial Corp.](#) Create *Body* objects in layouts and blocks by factory function `add_body()`. *ezdxf* will never interpret ACIS source code, don't ask me for this feature.

`Body.get_acis_data()`

Get the ACIS source code as a list of strings.

`Body.set_acis_data(test_lines)`

Set the ACIS source code as a list of strings **without** line endings.

`Body.edit_data()`

Context manager for ACIS text lines, returns `ModelerGeometryData`:

```
with body_entity.edit_data as data:
    # data.text_lines is a standard Python list
    # remove, append and modify ACIS source code
    data.text_lines = ['line 1', 'line 2', 'line 3'] # replaces the whole ACIS_
    ↪content (with invalid data)
```

ModelerGeometryData

ModelerGeometryData:

`ModelerGeometryData.text_lines`

ACIS data as list of strings. (read/write)

`ModelerGeometryData.__str__()`

Return concatenated `text_lines` as one string, lines are separated by `\n`.

Region

class `Region` (*Body*)

Introduced in DXF version R13 (AC1012), dxftype is REGION.

An object created by an ACIS based geometry kernel provided by the [Spatial Corp.](#) Create *Region* objects in layouts and blocks by factory function `add_region()`.

`Region.get_acis_data()`

Get the ACIS source code as a list of strings.

`Region.set_acis_data(test_lines)`

Set the ACIS source code as a list of strings **without** line endings.

`Region.edit_data()`

Context manager for ACIS text lines, returns `ModelerGeometryData`.

3DSolid

class `3DSolid`(*Body*)

Introduced in DXF version R13 (AC1012), dxftype is 3DSOLID.

A 3D object created by an ACIS based geometry kernel provided by the [Spatial Corp.](#) Create `3DSolid` objects in layouts and blocks by factory function `add_3dsolid()`.

DXF Attributes for 3DSolid

Common DXF Attributes for DXF R13 or later

`3DSolid.dxf.history`

Handle to history object, see: *Low Level Access to DXF entities*

3DSolid Methods

`3DSolid.get_acis_data()`

Get the ACIS source code as a list of strings.

`3DSolid.set_acis_data(test_lines)`

Set the ACIS source code as a list of strings **without** line endings.

`3DSolid.edit_data()`

Context manager for ACIS text lines, returns `ModelerGeometryData`.

Surface

class `Surface`(*Body*)

Introduced in DXF version R2007 (AC1021), dxftype is SURFACE.

A 3D object created by an ACIS based geometry kernel provided by the [Spatial Corp.](#) Create `Surface` objects in layouts and blocks by factory function `add_surface()`.

DXF Attributes for Surface

Common DXF Attributes for DXF R13 or later

`Surface.dxf.u_count`

Number of U isolines

`Surface.dxf.v_count`

Number of V isolines

Surface Methods

`Surface.get_acis_data()`

Get the ACIS source code as a list of strings.

`Surface.set_acis_data(test_lines)`

Set the ACIS source code as a list of strings **without** line endings.

`Surface.edit_data()`

Context manager for ACIS text lines, returns `ModelerGeometryData`.

ExtrudedSurface

class `ExtrudedSurface` (*Surface*)

Introduced in DXF version R2007 (AC1021), dxftype is EXTRUDED_SURFACE.

DXF Attributes for ExtrudedSurface

`ExtrudedSurface.dxf.class_id`

`ExtrudedSurface.dxf.sweep_vector`

`ExtrudedSurface.dxf.draft_angle`

`ExtrudedSurface.dxf.draft_start_distance`

`ExtrudedSurface.dxf.draft_end_distance`

`ExtrudedSurface.dxf.twist_angle`

`ExtrudedSurface.dxf.scale_factor`

`ExtrudedSurface.dxf.align_angle`

`ExtrudedSurface.dxf.solid`

`ExtrudedSurface.dxf.sweep_alignment_flags`

- 0 = No alignment
- 1 = Align sweep entity to path
- 2 = Translate sweep entity to path
- 3 = Translate path to sweep entity

`ExtrudedSurface.dxf.align_start`

`ExtrudedSurface.dxf.bank`

`ExtrudedSurface.dxf.base_point_set`

`ExtrudedSurface.dxf.sweep_entity_transform_computed`

`ExtrudedSurface.dxf.path_entity_transform_computed`

ExtrudedSurface.dxf.reference_vector_for_controlling_twist

ExtrudedSurface Methods

ExtrudedSurface.set_transformation_matrix_extruded_entity(*matrix*)

Parameters *matrix* – iterable of 16 numeric values.

ExtrudedSurface.get_transformation_matrix_extruded_entity()

Returns *Matrix44* object

ExtrudedSurface.set_sweep_entity_transformation_matrix(*matrix*)

Parameters *matrix* – iterable of 16 numeric values.

ExtrudedSurface.get_sweep_entity_transformation_matrix()

Returns *Matrix44* object

ExtrudedSurface.set_path_entity_transformation_matrix(*matrix*)

Parameters *matrix* – iterable of 16 numeric values.

ExtrudedSurface.get_path_entity_transformation_matrix()

Returns *Matrix44* object

LoftedSurface

class LoftedSurface (*Surface*)

Introduced in DXF version R2007 (AC1021), dxftype is LOFTEDSURFACE.

DXF Attributes for LoftedSurface

LoftedSurface.dxf.plane_normal_lofting_type

LoftedSurface.dxf.start_draft_angle

LoftedSurface.dxf.end_draft_angle

LoftedSurface.dxf.start_draft_magnitude

LoftedSurface.dxf.end_draft_magnitude

LoftedSurface.dxf.arc_length_parameterization

LoftedSurface.dxf.no_twist

LoftedSurface.dxf.align_direction

LoftedSurface.dxf.simple_surfaces

LoftedSurface.dxf.closed_surfaces

LoftedSurface.dxf.solid

LoftedSurface.dxf.ruled_surface

LoftedSurface.dxf.virtual_guide

LoftedSurface Methods

`LoftedSurface.set_transformation_matrix_lofted_entity(matrix)`

Parameters `matrix` – iterable of 16 numeric values.

`LoftedSurface.get_transformation_matrix_lofted_entity()`

Returns `Matrix44` object

RevolvedSurface

class `RevolvedSurface` (`Surface`)

Introduced in DXF version R2007 (AC1021), dxftype is REVOLVEDSURFACE.

DXF Attributes for RevolvedSurface

`RevolvedSurface.dxf.class_id`

`RevolvedSurface.dxf.axis_point`

`RevolvedSurface.dxf.axis_vector`

`RevolvedSurface.dxf.revolve_angle`

`RevolvedSurface.dxf.start_angle`

`RevolvedSurface.dxf.draft_angle`

`RevolvedSurface.dxf.start_draft_distance`

`RevolvedSurface.dxf.end_draft_distance`

`RevolvedSurface.dxf.twist_angle`

`RevolvedSurface.dxf.solid`

`RevolvedSurface.dxf.close_to_axis`

RevolvedSurface Methods

`RevolvedSurface.set_transformation_matrix_revolved_entity(matrix)`

Parameters `matrix` – iterable of 16 numeric values.

`RevolvedSurface.get_transformation_matrix_revolved_entity()`

Returns `Matrix44` object

SweptSurface

class `SweptSurface` (`Surface`)

Introduced in DXF version R2007 (AC1021), dxftype is SWEPTSURFACE.

DXF Attributes for SweptSurface

SweptSurface.dxf.**swept_entity_id**
SweptSurface.dxf.**path_entity_id**
SweptSurface.dxf.**draft_angle**
SweptSurface.dxf.**draft_start_distance**
SweptSurface.dxf.**draft_end_distance**
SweptSurface.dxf.**twist_angle**
SweptSurface.dxf.**scale_factor**
SweptSurface.dxf.**align_angle**
SweptSurface.dxf.**solid**
SweptSurface.dxf.**sweep_alignment**
SweptSurface.dxf.**align_start**
SweptSurface.dxf.**bank**
SweptSurface.dxf.**base_point_set**
SweptSurface.dxf.**sweep_entity_transform_computed**
SweptSurface.dxf.**path_entity_transform_computed**
SweptSurface.dxf.**reference_vector_for_controlling_twist**

SweptSurface Methods

SweptSurface.**set_transformation_matrix_sweep_entity**(*matrix*)

Parameters *matrix* – iterable of 16 numeric values.

SweptSurface.**get_transformation_matrix_sweep_entity**()

Returns *Matrix44* object

SweptSurface.**set_transformation_matrix_path_entity**(*matrix*)

Parameters *matrix* – iterable of 16 numeric values.

SweptSurface.**get_transformation_matrix_path_entity**()

Returns *Matrix44* object

SweptSurface.**set_sweep_entity_transformation_matrix**(*matrix*)

Parameters *matrix* – iterable of 16 numeric values.

SweptSurface.**get_sweep_entity_transformation_matrix**()

Returns *Matrix44* object

SweptSurface.**set_path_entity_transformation_matrix**(*matrix*)

Parameters *matrix* – iterable of 16 numeric values.

SweptSurface.**get_path_entity_transformation_matrix**()

Returns *Matrix44* object

Underlay

class Underlay (*GraphicEntity*)

Introduced in DXF version R13 (AC1012), dxfname is PDFUNDERLAY, DWFUNDERLAY or DGNUNDERLAY.

Add 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*. Create *Underlay* in layouts and blocks by factory function *add_underlay()*. The DXF standard supports three different fileformats: PDF, DWF (DWFx) and DGN. An Underlay can be clipped by a rectangle or a polygon path. The clipping coordinates are 2D OCS/ECS coordinates and in drawing units but without scaling.

DXF Attributes for Underlay

Common DXF Attributes for DXF R13 or later

`underlay.dxf.insert`

Insertion point, lower left corner of the image in *OCS*.

`underlay.dxf.scale_x`

scaling factor in x direction (float)

`underlay.dxf.scale_y`

scaling factor in y direction (float)

`underlay.dxf.scale_z`

scaling factor in z direction (float)

`underlay.dxf.rotation`

ccw rotation in degrees around the extrusion vector (float)

`underlay.dxf.extrusion`

extrusion vector (default=0, 0, 1)

`underlay.dxf.underlay_def`

Handle to the underlay definition entity, see *UnderlayDefinition*

`underlay.dxf.flags`

Underlay.dxf.flags	Value	Description
UNDERLAY_CLIPPING	1	clipping is on/off
UNDERLAY_ON	2	underlay is on/off
UNDERLAY_MONOCHROME	4	Monochrome
UNDERLAY_ADJUST_FOR_BACKGROUND	8	Adjust for background

`underlay.dxf.contrast`

Contrast value (20-100; default = 100)

`underlay.dxf.fade`

Fade value (0-80; default = 0)

Underlay Attributes

Underlay.**clipping**

True or False (read/write)

Underlay.**on**

True or False (read/write)

Underlay.**monochrome**

True or False (read/write)

Underlay.**adjust_for_background**

True or False (read/write)

Underlay.**scale**

Scaling (x, y, z) tuple (read/write)

Underlay Methods

Underlay.**get_boundary**()

Returns a list of vertices as pixel coordinates, just two values represent the lower left and the upper right corners of the clipping rectangle, more vertices describe a clipping polygon.

Underlay.**reset_boundary**()

Removes the clipping path.

Underlay.**set_boundary**(*vertices*)

Set boundary path to vertices. 2 points describe a rectangle (lower left and upper right corner), more than 2 points is a polygon as clipping path. Sets clipping state to 1.

Underlay.**get_underlay_def**()

returns the associated (PDF)DEFINITION entity. see *UnderlayDefinition*.

4.3.5 Blocks

Blocks Section

The *BlocksSection* class manages all block definitions of a drawing document.

class BlocksSection

BlocksSection.**__iter__**()

Iterate over all block definitions, yielding *BlockLayout* objects.

BlocksSection.**__contains__**(*entity*)

Test if *BlocksSection* contains the block definition *entity*, *entity* can be a block name as *str* or the *Block* definition itself.

BlocksSection.**__getitem__**(*name*)

Get the *Block* definition by *name*, raises *DXFKeyError* if no block *name* exists.

BlocksSection.**get**(*name*, *default=None*)

Get the *Block* definition by *name*, returns *default* if no block *name* exists.

`BlocksSection.new` (*name*, *base_point*=(0, 0), *dxfattribs*=None)

Create and add a new *Block*, *name* is the block-name, *base_point* is the insertion point of the block.

`BlocksSection.new_anonymous_block` (*type_char*='U', *base_point*=(0, 0))

Create and add a new anonymous *Block*, *type_char* is the block-type, 'base_point' is the insertion point of the block.

`BlocksSection.rename_block` (*old_name*, *new_name*)

Rename block 'old_name' in 'new_name'.

`BlockSection.delete_block` (*name*, *safe*=True)

Delete block *name*. If *safe* is True, check if block is still referenced.

Parameters

- **name** – block name (case insensitive)
- **safe** – check if block is still referenced

Raises

- **DXFKeyError** – if block *name* does not exist
- **DXFBlockInUseError** – if block *name* is still referenced, and *safe* is True

`BlockSection.delete_all_blocks` (*safe*=True)

Delete all blocks except layout blocks (model space or paper space).

Parameters **safe** – check if block is still referenced before deleting and ignore them if so

type_char	Anonymous Block Type
U	*U### anonymous blocks
E	*E### anonymous non-uniformly scaled blocks
X	*X### anonymous hatches
D	*D### anonymous dimensions
A	*A### anonymous groups
T	*T### anonymous block for ACAD_TABLE content

Block

class Block

Blocks are embedded into the *BlockLayout* object. The BLOCK entity is accessible by *BlockLayout.block*.

DXF Attributes for Block

Common DXF Attributes for DXF R12

Common DXF Attributes for DXF R13 or later

`Block.dxf.name`

Block name

`Block.dxf.name2`

The same block name a second time (meaning?)

`Block.dxf.base_point`

Defines the base point of the block, default=(0, 0, 0)

Block.dxf.flags

Block flags (bit-coded)

1	Anonymous block generated by hatching, associative dimensioning, other internal operations, or an application
2	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)
4	Block is an external reference (xref)
8	Block is an xref overlay
16	Block is externally dependent
32	This is a resolved external reference, or dependent of an external reference (ignored on input)
64	This definition is a referenced external reference (ignored on input)

Block.dxf.xref_path

Xref path name

Block.is_layout_block

True if block is a model space or paper space block definition.

Insert

class Insert

A block reference (DXF type: INSERT) with the possibility to append attributes (*Attrib*).

DXF Attributes for Insert

Common DXF Attributes for DXF R12

Common DXF Attributes for DXF R13 or later

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

Insert.dxf.name

Block name (str)

Insert.dxf.insert

Insertion point as (2D/3D Point in *OCS*)

Insert.dxf.xscale

Scale factor for x direction (float)

Insert.dxf.yscale

Scale factor for y direction (float)

Insert.dxf.zscale

Scale factor for z direction (float)

Insert.dxf.rotation

Rotation angle in degrees (float)

Insert.dxf.row_count

Count of repeated insertions in row direction (int)

Insert.dxf.row_spacing

Distance between two insert points in row direction (float)

`Insert.dxf.column_count`

Count of repeated insertions in column direction (int)

`Insert.dxf.column_spacing`

Distance between two insert points in column direction (float)

Insert Methods

`Insert.place` (*insert=None, scale=None, rotation=None*)

Place block reference as point *insert* with scaling and rotation. *scale* has to be a (x, y, z)-tuple and *rotation* a rotation angle in degrees. Parameters which are *None* will not be altered.

`Insert.grid` (*size=(1, 1), spacing=(1, 1)*)

Place block references in a grid layout with grid *size*=(rows, columns)-tuple and *spacing*=(row_spacing, column_spacing)-tuple. *spacing* is the distance from insertion point to insertion point.

`Insert.attrs` ()

Iterate over appended *Attrib* objects.

`Insert.has_attr` (*tag, search_const=False*)

Returns *True* if an attrib *tag* exists else *False*, for *search_const* doc see `Insert.get_attr` ().

`Insert.get_attr` (*tag, search_const=False*)

Get the appended *Attrib* object with `object.dxf.tag == tag`, returns *None* if not found. Some applications may not attach *Attrib*, which do represent constant values, set *search_const=True* and you get at least the associated *Attdef* entity.

`Insert.get_attr_text` (*tag, default=None, search_const=False*)

Get content text for attrib *tag* as string or return *default* if no attrib *tag* exists, for *search_const* doc see `Insert.get_attr` ().

`Insert.add_attr` (*tag, text, insert=(0, 0), attrs={}*)

Append an *Attrib* to the block reference. Returns an *Attrib* object.

Example for appending an attribute to an INSERT entity with none standard alignment:

```
insert_entity.add_attr("TAG", "example text").set_pos((3, 7), align='MIDDLE_CENTER')
```

`Insert.delete_attr` (*tag, ignore=False*)

Delete an *Attrib* from *Insert*. If *ignore* is *False*, an `DXFKeyError` exception is raised, if *Attrib tag* does not exist.

`Insert.delete_all_attrs` ()

Delete all attached *Attrib* entities.

AttDef

class Attdef

The *Attdef* entity is a place holder in the *Block* definition, which will be used to create an appended *Attrib* entity for an *Insert* entity.

DXF Attributes for AttDef

Common DXF Attributes for DXF R12

Common DXF Attributes for DXF R13 or later

`AttDef.dxf.text`

The default text prompted by CAD programs (str)

`AttDef.dxf.insert`

First alignment point of text (2D/3D Point in *OCS*), relevant for the adjustments LEFT, ALIGN and FIT.

`AttDef.dxf.tag`

Tag to identify the attribute (str)

`AttDef.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)

`AttDef.dxf.height`

Text height in drawing units (float), default is 1

`AttDef.dxf.rotation`

Text rotation in degrees (float), default is 0

`AttDef.dxf.oblique`

Text oblique angle (float), default is 0

`AttDef.dxf.style`

Text style name (str), default is STANDARD

`AttDef.dxf.width`

Width scale factor (float), default is 1

`AttDef.dxf.halign`

Horizontal alignment flag (int), use `Attdef.set_pos()` and `Attdef.set_align()`

`AttDef.dxf.valign`

Vertical alignment flag (int), use `Attdef.set_pos()` and `Attdef.set_align()`

`AttDef.dxf.text_generation_flag`

Text generation flags (int)

- 2 = text is backward (mirrored in X)
- 4 = text is upside down (mirrored in Y)

`AttDef.dxf.prompt`

Text prompted by CAD programs at placing a block reference containing this `Attdef`

`AttDef.dxf.field_length`

Just relevant to CAD programs for validating user input

AttDef Attributes

`Attdef.is_invisible`

(read/write) Attribute is invisible (does not appear).

`Attdef.is_const`

(read/write) This is a constant attribute.

`Attdef.is_verify`

(read/write) Verification is required on input of this attribute. (CAD application feature)

`Attdef.is_preset`

(read/write) No prompt during insertion. (CAD application feature)

AttDef Methods

`Attdef.get_pos()`

see method `Text.get_pos()`.

`Attdef.set_pos(p1, p2=None, align=None)`

see method `Text.set_pos()`.

`Attdef.get_align()`

see method `Text.get_align()`.

`Attdef.set_align(align='LEFT')`

see method `Text.set_align()`.

Attrib

class `Attrib`

The `Attrib` 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.

DXF Attributes for `Attrib`

Common DXF Attributes for DXF R12

Common DXF Attributes for DXF R13 or later

`Attrib.dxf.text`

Attribute content as text (str)

`Attrib.dxf.insert`

First alignment point of text (2D/3D Point in *OCS*), relevant for the adjustments LEFT, ALIGN and FIT.

`Attrib.dxf.tag`

Tag to identify the attribute (str)

`Attrib.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)

`Attrib.dxf.height`

Text height in drawing units (float), default is 1

`Attrib.dxf.rotation`

Text rotation in degrees (float), default is 0

`Attrib.dxf.oblique`

Text oblique angle (float), default is 0

`Attrib.dxf.style`

Text style name (str), default is STANDARD

`Attrib.dxf.width`

Width scale factor (float), default is 1

`Attrib.dxf.halign`

Horizontal alignment flag (int), use `Attrib.set_pos()` and `Attrib.set_align()`

`Attrib.dxf.valign`

Vertical alignment flag (int), use `Attrib.set_pos()` and `Attrib.set_align()`

`Attrib.dxf.text_generation_flag`

Text generation flags (int)

- 2 = text is backward (mirrored in X)
- 4 = text is upside down (mirrored in Y)

Attrib Attributes

`Attrib.is_invisibe`

(read/write) Attribute is invisible (does not appear).

`Attrib.is_const`

(read/write) This is a constant attribute.

`Attrib.is_verify`

(read/write) Verification is required on input of this attribute. (CAD application feature)

`Attrib.is_preset`

(read/write) No prompt during insertion. (CAD application feature)

Attrib Methods

`Attrib.get_pos()`

see method `Text.get_pos()`.

`Attrib.set_pos(p1, p2=None, align=None)`

see method `Text.set_pos()`.

Attrib.**get_align**()

see method *Text.get_align()*.

Attrib.**set_align**(align='LEFT')

see method *Text.set_align()*.

4.3.6 Groups

Group

A group is just a bunch of DXF entities tied together. All entities of a group has to be on the same layout (model space 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 an *Drawing.groups* of the class *Drawing*.

Group entities have to be in model space or any paper layout but not in a block definition!

class DXFGroup

DXFAttr	Version	Description
description	R13	group description (string)
unnamed	R13	1 for unnamed, 0 for named group (int)
selectable	R13	1 for selectable, 0 for not selectable group (int)

The group name is not stored in the GROUP entity, it is stored in the *DXFGroupTable* object.

DXFGroup.**__iter__**()

Iterate over all DXF entities in this group as instances of *GraphicEntity* or inherited (LINE, CIRCLE, ...).

DXFGroup.**__len__**()

Returns the count of DXF entities in this group.

DXFGroup.**__contains__**(item)

Returns *True* if item is in this group else *False*. *item* has to be a handle string or an object of type *GraphicEntity* or inherited.

DXFGroup.**handles**()

Generator over all entity handles in this group.

DXFGroup.**get_name**()

Get name of the group as *string*.

DXFGroup.**edit_data**()

Context manager which yields all the group entities as standard Python list:

```
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()
```

DXFGroup.**set_data**(entities)

Set *entities* as new group content, entities should be iterable and yields instances of *GraphicEntity* or inherited (LINE, CIRCLE, ...).

DXFGroup.**extend**(entities)

Append *entities* to group content, entities should be iterable and yields instances of *GraphicEntity* or inherited (LINE, CIRCLE, ...).

`DXFGroup.clear()`

Remove all entities from group.

`DXFGroup.remove_invalid_handles()`

Remove invalid handles from group. Invalid handles: deleted entities, entities in a block layout (but not implemented yet)

GroupTable

There only exists one group table in each drawing, which is accessible by the attribute `Drawing.groups`.

class DXFGroupTable

`DXFGroupTable.__iter__()`

Iterate over all existing groups as `(name, group)` tuples. `name` is the name of the group as `string` and `group` is an object of type `DXFGroup`.

`DXFGroupTable.groups()`

Generator over all existing groups, yields just objects of type `DXFGroup`.

`DXFGroupTable.__len__()`

Returns the count of DXF groups.

`DXFGroupTable.__contains__(name)`

Returns `True` if a group `name` exists else `False`.

`DXFGroupTable.get(name)`

Returns the group `name` as `DXFGroup` object. Raises `DXFKeyError` if no group `name` exists.

`DXFGroupTable.new(name=None, description="", selectable=1)`

Creates a new group, returns a `DXFGroup` object. If `name` is `None` an unnamed group is created, which has an automatically generated name like `'*Annnn'`. `description` is the group description as string and `selectable` defines if the group is selectable (`selectable=1`) or not (`selectable=0`).

`DXFGroupTable.delete(group)`

Delete `group`. `group` can be an object of type `DXFGroup` or a group name.

`DXFGroupTable.clear()`

Delete all groups.

`DXFGroupTable.cleanup()`

Removes invalid handles in all groups and empty groups.

4.3.7 Objects

Object Base Class

class DXFObject

Common base class for all non-graphical DXF objects.

`DXFObject.dxf`

(read only) The DXF attributes namespace, access DXF attributes by this attribute, like `entity.dxf.layer = 'MyLayer'`. Just the `dxf` attribute is read only, the DXF attributes are read- and writeable.

`DXFObject.drawing`

(read only) Get the associated drawing.

`DXFObject.dxfactory`

(read only) Get the associated DXF factory. (feature for experts)

`DXFObject.dxf_type()`
Get the DXF type string, like GEODATA for the geo data entity.

`DXFObject.copy()`
Deep copy of DXFObject with new handle. This is not a deep copy in the meaning of Python, because handle, link and owner is changed.

`DXFObject.get_dxf_attrib(key, default=DXFValueError)`
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.

`DXFObject.set_dxf_attrib(key, value)`

`DXFObject.del_dxf_attrib(key)`
Delete/remove DXF attribute *key*. Raises `AttributeError` if *key* isn't supported.

`DXFObject.dxf_attrib_exists(key)`
Returns `True` if DXF attrib *key* really exists else `False`. Raises `AttributeError` if *key* isn't supported

`DXFObject.supported_dxf_attrib(key)`
Returns `True` if DXF attrib *key* is supported by this entity else `False`. Does not grant that attrib *key* really exists.

`DXFObject.valid_dxf_attrib_names(key)`
Returns a list of supported DXF attribute names.

`DXFObject.dxfattribs()`
Create a dict() with all accessible DXF attributes and their value, not all data is accessible by dxf attributes like definition points of *LWPolyline* or *Spline*

`DXFObject.update_attribs(dxfattribs)`
Set DXF attributes by a dict() like `{'layer': 'test', 'color': 4}`.

`DXFObject.set_flag_state(flag, state=True, name='flags')`
Set binary coded *flag* of DXF attribute *name* to 1 (on) if *state* is `True`, set *flag* to 0 (off) if *state* is `False`.

`DXFObject.get_flag_state(flag, name='flags')`
Returns `True` if any *flag* of DXF attribute is 1 (on), else `False`. Always check just one flag state at the time.

Common DXF Object Attributes

`DXFObject.dxf.handle`
DXF handle (feature for experts)

`DXFObject.dxf.owner`
handle to owner, it's a `BLOCK_RECORD` entry (feature for experts)

ImageDef

class `ImageDef(DXFObject)`

Introduced in DXF version R13 (AC1012), `dxf_type` is `IMAGEDEF`.

ImageDef defines an image, which can be placed by the *Image* entity. Create *ImageDef* by the *Drawing* factory function `add_image_def()`.

DXF Attributes for ImageDef

- `ImageDef.dxf.filename`
Relative (to the DXF file) or absolute path to the image file as string
- `ImageDef.dxf.image_size`
Image size in pixel as (x, y) tuple
- `ImageDef.dxf.pixel_size`
Default size of one pixel in AutoCAD units (x, y) tuple
- `ImageDef.dxf.loaded`
Default = 1
- `ImageDef.dxf.resolution_units`
- 0 = No units
 - 2 = Centimeters
 - 5 = Inch
 - default is 0

UnderlayDefinition

class UnderlayDefinition (*DXFObject*)

Introduced in DXF version R13 (AC1012), dxftype is PDFDEFINITION, DWFDEFINITION and DGNDEFINITION.

UnderlayDefinition defines an underlay, which can be placed by the *Underlay* entity. Create *UnderlayDefinition* by the *Drawing* factory function *add_underlay_def()*.

DXF Attributes for UnderlayDefinition

- `UnderlayDefinition.dxf.filename`
Relative (to the DXF file) or absolute path to the image file as string
- `UnderlayDefinition.dxf.name`
defines what to display
- pdf: page number
 - dgn: 'default'
 - dwf: ???

GeoData

class GeoData (*DXFObject*)

Introduced in DXF version R2010 (AC1024), dxftype is GEODATA

The GEODATA entity is associated to the *Modelspace* object, create new geo data by *Modelspace.new_geodata()*, or get existing geo data by *Modelspace.get_geodata()*.

See also:

[using_geodata.py](#)

DXF Attributes for GeoData

GeoData.dxf.**version**

- 1 = 2009
- 2 = 2010

GeoData.dxf.**coordinate_type**

- 0 = unknown
- 1 = local grid
- 2 = projected grid
- 3 = geographic (latitude/longitude)

GeoData.dxf.**block_record**

Handle of host block table record.

GeoData.dxf.**design_point**

Reference point in WCS coordinates.

GeoData.dxf.**reference_point**

Reference point in coordinate system coordinates, valid only when coordinate type is Local Grid.

GeoData.dxf.**north_direction**

North direction as 2D vector.

GeoData.dxf.**horizontal_unit_scale**

Horizontal unit scale, factor which converts horizontal design coordinates to meters by multiplication.

GeoData.dxf.**vertical_unit_scale**

Vertical unit scale, factor which converts vertical design coordinates to meters by multiplication.

GeoData.dxf.**horizontal_units**

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

GeoData.dxf.**vertical_units**

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

GeoData.dxf.**up_direction**

Up direction as 3D vector.

GeoData.dxf.**scale_estimation_method**

- 1 = none
- 2 = user specified scale factor
- 3 = grid scale at reference point
- 4 = prismatic

GeoData.dxf.**sea_level_correction**

Bool flag specifying whether to do sea level correction.

GeoData.dxf.**user_scale_factor**

GeoData.dxf.**sea_level_elevation**

GeoData.dxf.**coordinate_projection_radius**

GeoData.dxf.**geo_rss_tag**

GeoData.dxf.**observation_from_tag**

GeoData.dxf.**observation_to_tag**

GeoData.dxf.**mesh_faces_count**

GeoData Methods

GeoData.**get_coordinate_system_definition**()

Returns Coordinate system definition string (always a XML string?)

GeoData.**set_coordinate_system_definition**(*text*)

GeoData.**get_mesh_data**()

Returns mesh as list of vertices and list of faces. Each vertex entry is a 2-tuple of source and target point, vertices are 2D points. Each face entry is a 3-tuple of vertex indices (0 based).

Returns tuple (vertices, faces)

GeoData.**set_mesh_data**(*vertices=None, faces=None*)

Each vertex entry is a 2-tuple of source and target point, all vertices are 2D points. Each face entry is a 3-tuple of vertex indices (0 based), faces are optional.

4.3.8 Importer

Import data from other DXF drawings

class Importer

Import definitions and entities from other DXF drawings.

- can import line-, text-, dimension-styles and layer-definitions
- can import block-definitions
- can import entities from model-space
- **can't** import layouts
- **can't** import entities from layouts

Compatible Drawings

- It is always possible to copy from older to newer versions (except R12).
- It is possible to copy an entity from a newer to an older versions, if the entity is defined for both versions (like LINE, CIRCLE, ...), but this can not be granted by default. Enable this feature by *Importer(s, t, strict_mode=False)*.

Incompatible Drawings

The basic DXF structure has been changed with version AC1012 (AutoCAD R13):

- **can't** copy from R12 to newer versions, it's possible if *strict_mode=False*, but the target drawing is *invalid*.
- **can't** copy from newer versions to R12, it's possible if *strict_mode=False*, but the target drawing is *invalid*.

Importer.**__init__**(*source, target, strict_mode=True*)

Parameters

- **source** – source drawing of type *Drawing*

- **target** – target drawing of type *Drawing*
- **strict_mode** (*bool*) – import is only possible, if the drawings are compatible.

Now you can import DXF tables, like layer definitions and dimension style definitions or block definitions from the blocks section or DXF entities from the model-space.

First create an *Importer* object:

```
import ezdxf

source_drawing = ezdxf.readfile("Source_DXF_Drawing.dxf")
target_drawing = ezdxf.new(dxfversion=source_drawing.dxfversion)
importer = ezdxf.Importer(source_drawing, target_drawing)
```

Import Tables

Import line-, text-, dimension-styles and layer-definitions from other DXF drawing.

`Importer.import_tables` (*query='*', conflict='discard'*)

Import all tables listed by the query string, * means all tables. Valid table names are layers, linetypes, appids, dimstyles, styles, ucs, views, viewports and block_records.

`Importer.import_table` (*name, query='*', conflict='discard'*)

Import table entries from a specific table, the query string specifies the entries to import, * means all table entries.

Parameters

- **query** (*str*) – is a *Name Query String*
 - **conflict** (*str*) – discard|replace
- discard: already existing entries will be preserved
 - replace: already existing entries will be replaced by entries from the source drawing

Import Block Definitions

Import block-definitions from other DXF drawings.

`Importer.import_blocks` (*query='*', conflict='discard'*)

Import block definitions, the query string specifies the blocks to import, * means all blocks.

Parameters

- **query** (*str*) – is a *Name Query String*
 - **conflict** (*str*) – discard|replace|rename
- discard: already existing blocks will be preserved
 - replace: already existing blocks will be replaced by blocks from the source drawing
 - rename: the imported block gets a new name, existing references in the source drawing will be resolved if possible. Block references in the model-space will be resolved, if they are imported AFTER importing the block definitions.

Import Model-Space Entities

Import entities from model-space of other DXF drawings.

`Importer.import_modelspace_entities(query='*')`

Import DXF entities from source model-space to the target model-space, select DXF types to import by the query string, * means all DXF types. If called *after* the `import_blocks()` method, references to renamed blocks will be resolved.

Parameters `query` (*str*) – is an *Entity Query String*

Additional Methods

`Importer.is_compatible()`

True if drawings are compatible, else *False*.

`Importer.import_all(table_conflict='discard', block_conflict='discard')`

Import all tables, block-definitions and entities from model-space.

4.3.9 Data Query

Name Query String

A name query string is just a standard regular expression see: <http://docs.python.org/3/library/re.html>

A '\$' will be appended to the query string.

For general usage of the query features see the tutorial: *Tutorial for Getting Data from DXF Files*

Entity Query String

```
QueryString := EntityQuery ("[" AttribQuery "]" "i"?) *
```

The query string is the combination of two queries, first the required entity query and second the *optional* attribute query, enclosed in square brackets, append 'i' after the closing square bracket to ignore case for strings.

Entity Query

The entity query is a whitespace separated list of DXF entity names or the special name '*'. Where '*' means all DXF entities, all other DXF names have to be uppercase.

Attribute Query

The *optional* attribute query is a boolean expression, supported operators are:

- not (!): !term is true, if term is false
- and (&): term & term is true, if both terms are true
- or (|): term | term is true, if one term is true
- and arbitrary nested round brackets
- append (i) after the closing square bracket to ignore case for strings

Attribute selection is a term: “name comparator value”, where name is a DXF entity attribute in lowercase, value is a integer, float or double quoted string, valid comparators are:

- "==" equal “value”
- "!=" not equal “value”
- "<" lower than “value”
- "<=" lower or equal than “value”
- ">" greater than “value”
- ">=" greater or equal than “value”
- "?" match regular expression “value”
- "!?" does not match regular expression “value”

Query Result

The `EntityQuery` class is the return type of all `query()` methods. `EntityQuery` contains all DXF entities of the source collection, which matches one name of the entity query AND the whole attribute query. If a DXF entity does not have or support a required attribute, the corresponding attribute search term is false.

examples:

```
'LINE[text ? ".*"]' is always empty, because the LINE entity has no text attribute.
'LINE CIRCLE[layer=="construction"]' => all LINE and CIRCLE entities on layer
↳ "construction"
'*[!(layer=="construction" & color<7)]' => all entities except those on layer ==
↳ "construction" and color < 7
'*[layer=="construction"]i' => (ignore case) all entities with layer == "construction
↳ " | "Construction" | "ConStruction" ...
```

EntityQuery Class

class `EntityQuery` (*Sequence*)

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 sequence methods and protocols. ([Python Sequence Docs](#))

`EntityQuery.__init__` (*entities, query='**)

Setup container with entities matching the initial query.

Parameters

- **entities** – sequence of wrapped DXF entities (at least `GraphicEntity` class)
- **query** (*str*) – entity query string

`EntityQuery.extend` (*entities, query='*', unique=True*)

Extent the query container by entities matching an additional query.

`EntityQuery.remove` (*query='**)

Remove all entities from result container matching this additional query.

`EntityQuery.query` (*query='**)

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

`EntityQuery.groupby(dxattrib="", key=None)`

Returns a mapping of this result container, where entities are grouped by a `dxattrib` or a key function.

Parameters

- **dxattrib** (*str*) – grouping DXF attribute like ‘layer’
- **key** (*function*) – 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, query='*')`

Start a new query based on a sequence *entities*. The sequence *entities* has to provide the Python iterator protocol and has to yield at least subclasses of `GenericWrapper` or better `GraphicEntity`. Returns an object of type `EntityQuery`.

4.3.10 Fast DXF R12 File/Stream Writer

Fast DXF R12 File/Stream Writer

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 _ _ _ _ _
- DASHDX2 _____
- DASHED2 _ _ _ _ _
- PHANTOM _____
- PHANTOMX2 _____
- PHANTOM2 _____
- DASHDOT _ . _ . _ . _ . _ . _ . _ . _ . _
- DASHDOTX2 _____ . _____ . _____
- DASHDOT2 _ . _ . _ . _ . _ . _ . _ . _ . _

(continued from previous page)

```

]
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='ARIAL_NARROW')

```

Reference

r12writer (*stream*, *fixed_tables=False*)

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.

class R12FastStreamWriter

Fast stream writer to create simple DXF R12 drawings.

`R12FastStreamWriter.__init__` (*stream*, *fixed_tables=False*)

Constructor, *stream* should be a file like object with a *write* method. 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.

`R12FastStreamWriter.close` ()

Writes the DXF tail. Call is not necessary when using the context manager `r12writer()`.

`R12FastStreamWriter.add_line` (*start*, *end*, *layer="0"*, *color=None*, *linetype=None*)

Add a LINE entity from *start* to *end*.

Parameters

- **start** – start vertex 2d/3d vertex as (x, y [,z]) tuple
- **end** – end vertex 2d/3d 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 ACI (AutoCAD Color Index) as integer 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 is written some predefined line types are available, else line type is always *ByLayer*, which is always *Continuous* without a LAYERS table.

`R12FastStreamWriter.add_circle` (*center*, *radius*, *layer="0"*, *color=None*, *linetype=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 ACI see `add_line()`
- **linetype** – line type as string see `add_line()`

`R12FastStreamWriter.add_arc` (*center, radius, start=0, end=360, layer="0", color=None, linetype=None*)

Add an ARC entity. The arc goes counter clockwise from *start* angle to *end* angle.

Parameters

- **center** – center point of arc as (x, y) tuple
- **radius** – arc radius as float
- **start** – arc start angle in degrees as float (360 degree = circle)
- **end** – arc end angle in degrees as float
- **layer** – layer name as string, see `add_line()`
- **color** – color as ACI, see `add_line()`
- **linetype** – line type as string, see `add_line()`

`R12FastStreamWriter.add_point` (*location, layer="0", color=None, linetype=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 ACI, see `add_line()`
- **linetype** – line type as string, see `add_line()`

`R12FastStreamWriter.add_3dface` (*vertices, invisible=0, layer="0", color=None, linetype=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** – list 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 ACI, see `add_line()`
- **linetype** – line type as string, see `add_line()`

`R12FastStreamWriter.add_solid` (*vertices, layer="0", color=None, linetype=None*)

Add a SOLID entity. SOLID is a solid filled area with 3 or 4 edges and SOLID is 2d entity.

Parameters

- **vertices** – list of 3 or 4 (x, y [,z]) tuples, z axis will be ignored.
- **layer** – layer name as string, see `add_line()`
- **color** – color as ACI, see `add_line()`
- **linetype** – line type as string, see `add_line()`

`R12FastStreamWriter.add_polyline` (*vertices, layer="0", color=None, linetype=None*)

Add a POLYLINE entity. The first vertex (axis count) defines, if the POLYLINE is 2d or 3d.

Parameters

- **vertices** – list of (x, y [,z]) tuples, handles generators without building a temporary lists.

- **layer** – layer name as string, see `add_line()`
- **color** – color as ACI, see `add_line()`
- **linetype** – line type as string, see `add_line()`

`R12FastStreamWriter.add_text(text, insert=(0, 0), height=1., width=1., align="LEFT", rotation=0., oblique=0., style='STANDARD', layer="0", color=None)`

Add a one line TEXT entity.

Parameters

- **text** – the text as string
- **insert** – insert point 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 (360 degree = circle)
- **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 ACI, see `add_line()`

Vert/Horiz	Left	Center	Right
Top	TOP_LEFT	TOP_CENTER	TOP_RIGHT
Middle	MIDDLE_LEFT	MIDDLE_CENTER	MIDDLE_RIGHT
Bottom	BOTTOM_LEFT	BOTTOM_CENTER	BOTTOM_RIGHT
Baseline	LEFT	CENTER	RIGHT

The special alignments ALIGNED and FIT are not available.

4.3.11 Algebra Utilities

This utilities located in module `ezdxf.algebra`:

```
from ezdxf.algebra import Vector
```

Functions

`ezdxf.algebra.is_close(a, b)`
 Returns True if value is close to value b, uses `math.isclose(a, b, abs_tol=1e-9)` for Python 3, and emulates this function for Python 2.7.

`ezdxf.algebra.is_close_points(p1, p2)`
 Returns True if all axis of p1 and p2 are close.

`ezdxf.algebra.bspline_control_frame(fit_points, degree=3, method='distance', power=.5)`
 Generates the control points for the B-spline control frame by [Curve Global Interpolation](#). Given are the fit points and the degree of the B-spline. The function provides 3 methods for generating the parameter vector t:

1. `method = uniform`, creates a uniform `t` vector, from 0 to 1 evenly spaced; see [uniform](#) method
2. `method = distance`, creates a `t` vector with values proportional to the fit point distances, see [chord length](#) method
3. `method = centripetal`, creates a `t` vector with values proportional to the fit point distances^{power}; see [centripetal](#) method

Parameters

- **fit_points** – fit points of B-spline, as list of (x, y[, z]) tuples
- **degree** – degree of B-spline
- **method** – calculation method for parameter vector `t`
- **power** – power for centripetal method

Returns a *BSpline* object, with *BSpline.control_points* containing the calculated control points, also *BSpline.knot_values()* returns the used `knot` values.

Bulge Related Functions

`ezdxf.algebra.bulge_center` (*start_point*, *end_point*, *bulge*)
Calculate center of arc described by the given bulge parameters.

Parameters

- **start_point** – start point as (x, y) tuple
- **end_point** – end point as (x, y) tuple
- **bulge** – bulge value as float

Returns arc center as *Vector*

`ezdxf.algebra.bulge_radius` (*start_point*, *end_point*, *bulge*)
Calculate radius of arc defined by the given bulge parameters.

Parameters

- **start_point** – start point as (x, y) tuple
- **end_point** – end point as (x, y) tuple
- **bulge** – bulge value as float

Returns arc radius as float

`ezdxf.algebra.arc_to_bulge` (*center*, *start_angle*, *end_angle*, *radius*)
Calculate bulge parameters from arc parameters.

Parameters

- **center** – circle center point as (x, y) tuple
- **start_angle** – start angle in radians
- **end_angle** – end angle in radians
- **radius** – circle radius

Returns (*start_point*, *end_point*, *bulge*)

`ezdxf.algebra.bulge_to_arc` (*start_point*, *end_point*, *bulge*)
Calculate arc parameters from bulge parameters.

Parameters

- **start_point** – start point as (x, y) tuple
- **end_point** – end point as (x, y) tuple
- **bulge** – bulge value as float

Returns (center, start_angle, end_angle, radius)

`ezdxf.algebra.bulge_3_points` (*start_point, end_point, point*)
Calculate bulge value defined by three points.

Parameters

- **start_point** – start point of arc
- **end_point** – end point of arc
- **point** – arbitrary point on arc

Returns bulge value as float

This classes located in module `ezdxf.algebra`:

```
from ezdxf.algebra import OCS, UCS
```

OCS Class

```
class ezdxf.algebra.OCS
```

```
OCS.__init__(extrusion=(0, 0, 1))
```

Establish an Object Coordinate System for a given extrusion vector.

```
OCS.to_wcs(point)
```

Calculate world coordinates for point in object coordinates.

```
OCS.points_to_wcs(points)
```

Translate multiple object coordinates into world coordinates (generator).

```
OCS.from_wcs(point)
```

Calculate object coordinates for point in world coordinates.

```
OCS.points_from_wcs(points)
```

Translate multiple world coordinates into object coordinates (generator).

See also:

[OCS](#)

UCS Class

```
class ezdxf.algebra.UCS
```

```
UCS.__init__(origin=(0, 0, 0), ux=None, uy=None, uz=None)
```

Establish an User Coordinate System. The UCS is defined by the origin and two unit vectors for the x-, y- or z-axis, all axis n WCS. The missing axis is the cross product of the given axis.

If x- and y-axis are None: $ux=(1, 0, 0)$, $uy=(0, 1, 0)$, $uz=(0, 0, 1)$.

Normalization of unit vectors is not required.

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*

`UCS.to_wcs` (*point*)

Calculate world coordinates for point in UCS coordinates.

`UCS.points_to_wcs` (*points*)

Translate multiple user coordinates into world coordinates (generator).

`UCS.to_ocs` (*point*)

Calculate *OCS* coordinates for point in UCS coordinates.

OCS is defined by the z-axis of the UCS.

`UCS.points_from_wcs` (*points*)

Translate multiple user coordinates into *OCS* coordinates (generator).

OCS is defined by the z-axis of the UCS.

`UCS.from_wcs` (*point*)

Calculate UCS coordinates for point in world coordinates.

`UCS.points_from_wcs` (*points*)

Translate multiple world coordinates into user coordinates (generator).

`UCS.from_x_axis_and_point_in_xy` (*origin, axis, point*)

Returns an new *UCS* defined by the origin, the x-axis vector and an arbitrary point in the xy-plane. (static method)

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*

`UCS.from_x_axis_and_point_in_xz` (*origin, axis, point*)

Returns a new *UCS* defined by the origin, the x-axis vector and an arbitrary point in the xz-plane. (static method)

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*

`UCS.from_y_axis_and_point_in_xy` (*origin, axis, point*)

Returns a new *UCS* defined by the origin, the y-axis vector and an arbitrary point in the xy-plane. (static method)

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*

`UCS.from_y_axis_and_point_in_yz` (*origin, axis, point*)

Returns an new `UCS` defined by the origin, the y-axis vector and an arbitrary point in the yz-plane. (static method)

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`

`UCS.from_z_axis_and_point_in_xz` (*origin, axis, point*)

Returns an new `UCS` defined by the origin, the z-axis vector and an arbitrary point in the xz-plane. (static method)

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`

`UCS.from_z_axis_and_point_in_yz` (*origin, axis, point*)

Returns an new `UCS` defined by the origin, the z-axis vector and an arbitrary point in the yz-plane. (static method)

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`

See also:

`UCS`

This class located in module `ezdxf.algebra`:

```
from ezdxf.algebra import Vector
```

Vector

class `ezdxf.algebra.Vector`

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:

```
v1 = Vector(1, 2, 3)
v2 = v1
v2.z = 7 # this is not possible, raises AttributeError
v2 = Vector(v2.x, v2.y, 7) # this creates a new Vector() object
assert v1.z == 3 # and v1 remains unchanged
```

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)`

- `Vecotr(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:

```
v = Vector(1, 2, 3)
v + (1, 2, 3) == Vector(2, 4, 6)
(1, 2, 3) + v == Vector(2, 4, 6)
v - (1, 2, 3) == Vector(0, 0, 0)
(1, 2, 3) - v == Vector(0, 0, 0)
v * 3 == Vector(3, 6, 9)
3 * v == Vector(3, 6, 9)
Vector(3, 6, 9) / 3 == Vector(1, 2, 3)
-Vector(1, 2, 3) == (-1, -2, -3)
```

Comparison between vectors and vectors to tuples is supported:

```
Vector(1, 2, 3) < Vector(2, 2, 2)
(1, 2, 3) < tuple(Vector(2, 2, 2)) # conversion necessary
Vector(1, 2, 3) == (1, 2, 3)

bool(Vector(1, 2, 3)) is True
bool(Vector(0, 0, 0)) is False
```

Vector Attributes

`Vector.x`

`Vector.y`

`Vector.z`

`Vector.xy`

Returns `Vector(x, y, 0)`

`Vector.xyz`

Returns `(x, y, z)` tuple

`Vector.magnitude`

Returns length of vector

`Vector.magnitude_square`

Returns square length of vector

`Vector.is_null`

Returns True for `Vector(0, 0, 0)` else False

`Vector.spatial_angle_rad`

Returns spatial angle between vector and x-axis in radians

`Vector.spatial_angle_deg`

Returns spatial angle between vector and x-axis in degrees

`Vector.angle_rad`

Returns angle of vector in the xy-plane in radians.

`Vector.angle_deg`

Returns angle of vector in the xy-plane in degrees.

Vector Methods

`Vector.replace` (*x=None, y=None, z=None*)

Return new `Vector()` with replaced components != None.

`Vector.generate` (*items*)

Static method returns generator of `Vector()` objects created from items.

`Vector.list` (*items*)

Static method returns list of `Vector()` objects created from items.

`Vector.from_rad_angle` (*angle, length=1.*)

Static method returns `Vector()` from angle scaled by length, angle in radians.

`Vector.from_deg_angle` (*angle, length=1.*)

Static method returns `Vector()` from angle scaled by length, angle in degree.

`Vector.__str__` ()

Return (*x, y, z*) as string.

`Vector.__repr__` ()

Return `Vector(x, y, z)` as string.

`Vector.__len__` ()

Returns always 3

`Vector.__hash__` ()

`Vector.copy` ()

Returns copy of vector.

`Vector.__copy__` ()

Support for `copy.copy()`.

`Vector.__deepcopy__` (*memodict*)

Support for `copy.deepcopy()`.

`Vector.__getitem__` (*index*)

Support for indexing `v[0] == v.x; v[1] == v.y; v[2] == v.z;`

`Vector.__iter__` ()

Support for the Python iterator protocol.

`Vector.__abs__` ()

Returns length (magnitude) of vector.

`Vector.orthogonal` (*ccw=True*)

Returns orthogonal 2D vector, z value is unchanged.

Parameters `ccw` – counter clockwise if True else clockwise

`Vector.lerp` (*other, factor=.5*)

Linear interpolation between vector and other, returns new `Vector()` object.

Parameters

- **other** – target vector/point
- **factor** – interpolation factor (0==self, 1=other, 0.5=mid point)

`Vector.project` (*other*)

Project vector other onto self, returns new `Vector()` object.

`Vector.normalize` (*length=1*)

Returns new normalized `Vector()` object, optional scaled by length.

`Vector.reversed()`
Returns -vector as new `Vector()` object

`Vector.__neg__()`
Returns -vector as new `Vector()` object

`Vector.__bool__()`
Returns True if vector != (0, 0, 0)

`Vector.__eq__(other)`

`Vector.__lt__(other)`

`Vector.__add__(other)`

`Vector.__radd__(other)`

`Vector.__sub__(other)`

`Vector.__rsub__(other)`

`Vector.__mul__(other)`

`Vector.__rmul__(other)`

`Vector.__truediv__(other)`

`Vector.__div__(other)`

`Vector.__rtruediv__(other)`

`Vector.__rdiv__(other)`

`Vector.dot(other)`
Returns 'dot' product of vector . other.

`Vector.cross(other)`
Returns 'cross' product of vector x other

`Vector.distance(other)`
Returns distance between vector and other.

`Vector.angle_between(other)`
Returns angle between vector and other in th xy-plane in radians. +angle is counter clockwise orientation.

`Vector.rot_z_rad(angle)`
Return rotated vector around z axis, angle in radians.

`Vector.rot_z_deg(angle)`
Return rotated vector around z axis, angle in degrees.

This utilities located in module `ezdxf.algebra`:

```
from ezdxf.algebra import Matrix44
```

Matrix44

`class ezdxf.algebra.Matrix44`

This is a pure Python implementation for 4x4 transformation matrices, to avoid dependency to big numerical packages like numpy, and before binary wheels, installation of these packages wasn't always easy on Windows.

Matrix44 initialization:

- `Matrix44()` is 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.

`Matrix44.set(*args)`

Reset 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.

`Matrix44.__repr__()`

Returns the representation string of the matrix:

```
Matrix44((col0, col1, col2, col3), (...), (...), (...))
```

`Matrix44.get_row(row)`

Get row as list of of four float values.

`Matrix44.set_row(row, values)`

Sets the values in a row.

Parameters

- **row** – row index [0..3]
- **values** – four column values as iterable.

`Matrix44.get_col(col)`

Get column as list of of four float values.

`Matrix44.set_col(col, values)`

Sets the values in a column.

Parameters

- **col** – column index [0..3]
- **values** – four column values as iterable.

`Matrix44.copy()`

`Matrix44.__copy__()`

`Matrix44.scale(sx, sy=None, sz=None)`

Class method returns a scaling transformation matrix. If `sy` is `None`, `sy = sx`, and if `sz` is `None` `sz = sx`.

`Matrix44.translate(x, y, z)`

Class method returns a translation matrix to (x, y, z).

`Matrix44.x_rotate(angle)`

Class method returns a rotation matrix about the x-axis.

Parameters **angle** – rotation angle in radians

`Matrix44.y_rotate(angle)`

Class method returns a rotation matrix about the y-axis.

Parameters **angle** – rotation angle in radians

`Matrix44.z_rotate(angle)`

Class method returns a rotation matrix about the z-axis.

param angle rotation angle in radians

`Matrix44.axis_rotate` (*axis, angle*)

Class method returns a rotation matrix about an arbitrary axis.

Parameters

- **axis** – rotation axis as (x, y, z) tuple
- **angle** – rotation angle in radians

`Matrix44.xyz_rotate` (*angle_x, angle_y, angle_z*)

Class method 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

`Matrix44.perspective_projection` (*left, right, top, bottom, near, far*)

Class method 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

`Matrix44.perspective_projection_fov` (*fov, aspect, near, far*)

Class method 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

`Matrix44.chain` (**matrices*)

Compose a transformation matrix from one or more matrices.

`Matrix44.__setitem__` (*coord, value*)

Set (row, column) element.

`Matrix44.__getitem__` (*coord*)

Get (row, column) element.

`Matrix44.__iter__` ()

Iterates over all matrix values.

`Matrix44.__mul__` (*other*)

Returns a new matrix as result of the matrix multiplication with another matrix.

`Matrix44.__imul__` (*other*)

Inplace multiplication with another matrix.

`Matrix44.fast_mul (other)`

Multiplies this matrix with other matrix inplace.

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 `__imul__()`.

`Matrix44.rows ()`

Iterate over rows as 4-tuples.

`Matrix44.columns ()`

Iterate over columns as 4-tuples.

`Matrix44.transform (vector)`

Transforms a 3d vector and return the result as a tuple.

`Matrix44.transform_vectors (vectors)`

Returns a list of transformed vectors.

`Matrix44.transpose ()`

Swaps the rows for columns inplace.

`Matrix44.get_transpose ()`

Returns a new transposed matrix.

`Matrix44.determinant ()`

Returns determinant.

`Matrix44.inverse ()`

Returns the inverse of the matrix.

Raises `ZeroDivisionError` – if matrix has no inverse.

This classes located in module `ezdxf.algebra`:

```
from ezdxf.algebra import BSpline
```

BSpline

class `ezdxf.algebra.BSpline`

Calculate the vertices of a B-spline curve, using an uniform open `knot` vector (`clamped curve`).

`BSpline.control_points`

Control points as list of `Vector` objects

`BSpline.count`

Count of control points, (n + 1 in math definition).

`BSpline.order`

Order of B-spline = degree + 1

`BSpline.degree`

Degree (p) of B-spline = order - 1

`BSpline.max_t`

Max `knot` value.

`BSpline.knot_values ()`

Returns a list of `knot` values as floats, the knot vector always has order+count values (n + p + 2 in math definition)

`BSpline.basis_values (t)`

Returns the `basis` vector for position t.

`BSpline.approximate` (*segments*)

Approximates the whole B-spline from 0 to `max_t`, by line segments as a list of vertices, vertices count = `segments + 1`

`BSpline.point` (*t*)

Returns the B-spline vertex at position *t* as (x, y[, z]) tuple.

BSplineU

class `ezdxf.algebra.BSpline` (*BSpline*)

Calculate the points of a B-spline curve, uniform (periodic) knot vector (open curve).

BSplineClosed

class `ezdxf.algebra.BSplineClosed` (*BSplineU*)

Calculate the points of a closed uniform B-spline curve (closed curve).

DBSpline

class `ezdxf.algebra.DBspline` (*BSpline*)

Calculate points and derivative of a B-spline curve, using an uniform open knot vector (clamped curve).

`DBspline.point` (*t*)

Returns the B-spline vertex, 1. derivative and 2. derivative at position *t* as tuple (vertex, d1, d2), each value is a (x, y, z) tuple.

DBSplineU

class `ezdxf.algebra.DBsplineU` (*DBspline*)

Calculate points and derivative of a B-spline curve, uniform (periodic) knot vector (open curve).

DBSplineClosed

class `ezdxf.algebra.DBsplineClosed` (*DBsplineU*)

Calculate the points and derivative of a closed uniform B-spline curve (closed curve).

This class located in module `ezdxf.algebra`:

```
from ezdxf.algebra import Arc
```

Arc

class `ezdxf.algebra.Arc`

This is a helper class to create parameters for the DXF `Arc` class.

`Arc.center`

Center point as *Vector*

`Arc.radius`

Arc radius

`Arc.start_angle`

Start angle of arc in degrees.

`Arc.start_angle_rad`
Start angle of arc in radians.

`Arc.end_angle`
End angle of arc in degrees.

`Arc.end_angle_rad`
End angle of arc in radians.

Class Methods

`Arc.from_2p_angle` (*start_point, end_point, angle, ccw=True*)
Create arc from two points and enclosing angle. Arc goes by default in counter clockwise orientation from *start_point* to *end_point*, can be changed by *ccw =False*. Z-axis of *start_point* and *end_point* has to be 0 if given.

Parameters

- **start_point** – start point as (x, y [,z]) tuple
- **end_point** – end point as (x, y [,z]) tuple
- **angle** (*float*) – enclosing angle in degrees
- **ccw** (*bool*) – counter clockwise orientation

Return new *Arc*

`Arc.from_2p_radius` (*start_point, end_point, radius, ccw=True, center_is_left=True*)
Create arc from two points and arc radius. Arc goes by default in counter clockwise orientation from *start_point* to *end_point*, can be changed by *ccw =False*. Z-axis of *start_point* and *end_point* has to be 0 if given.

The parameter *center_is_left* defines if the center of the arc is left or right of the line *start point -> end point*. Parameter *ccw =False* swaps start- and end point, which inverts the meaning of *center_is_left*.

Parameters

- **start_point** – start point as (x, y [,z]) tuple
- **end_point** – end point as (x, y [,z]) tuple
- **radius** (*float*) – arc radius
- **ccw** (*bool*) – counter clockwise orientation
- **center_is_left** (*bool*) – center point of arc is left of line *start point -> end point* if True, else on the right side of this line

Return new *Arc*

`Arc.from_3p` (*start_point, end_point, def_point, ccw=True*)
Create arc from three points. Arc goes by default in counter clockwise orientation from *start_point* to *end_point*, can be changed by *ccw =False*. Z-axis of *start_point*, *end_point* and *def_point* has to be 0 if given.

Parameters

- **start_point** – start point as (x, y [,z]) tuple
- **end_point** – end point as (x, y [,z]) tuple
- **def_point** – additional definition point as (x, y [,z]) tuple
- **ccw** (*bool*) – counter clockwise orientation

Return new *Arc*

Methods

`Arc.add_to_layout` (*layout*, *ucs=None*, *dxfattribs=None*)

Add arc as DXF entity to a layout.

Supports 3D arcs by using an *UCS*. An arc is always defined in the xy-plane, by using an arbitrary *UCS*, the arc can be placed in 3D space, automatically *OCS* transformation included.

Parameters

- **layout** – destination layout (model space, paper space or block)
- **ucs** – *UCS* definition for arc properties transformation to *OCS*, *None* for 2D arcs
- **dxfattribs** (*dict*) – usual DXF attributes supported by *Arc*

Return DXF *Arc* object

4.3.12 Tag Data Structures

Tag Data Structures

Required DXF tag interface:

- property code: group code as int
- property value: tag value of unspecific type
- method `dxfst()`: returns the DXF string
- method `clone()`: returns a deep copy of tag

DXFTag

The *DXFTag* is the basic DXF data type, this data type is per definition immutable (but not per implementation for sake of speed)

class `ezdxf.lldxf.DXFTag`

`DXFTag.code`

DXF group code (read only), interface definition

`DXFTag.value`

DXF tag value (read only)

`DXFTag.__str__()`

`DXFTag.__repr__()`

`DXFTag.__getitem__()`

Support for indexing.

`DXFTag.__iter__()`

Support for iterator protocol.

`DXFTag.dxfstr()`

Returns the DXF string e.g. ‘0nLINEn’

`DXFTag.clone()`

Returns a clone of itself, this method is necessary for the more complex (and not immutable) DXF tag types.

DXFBinaryTag

Represents binary data compact as binary strings. This tag is immutable.

```
class ezdxf.lldxf.DXFBinaryTag (DXFTag)
```

DXFVertex

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 *mutable*.

```
class ezdxf.lldxf.DXFVertex (DXFTag)
```

```
DXFVertex.code
```

DXF group code for the x-component (read only)

```
DXFVertex.value
```

x, y[, z] coordinates as array.array('d') object (read/write)

```
DXFVertex.__str__()
```

```
DXFVertex.__repr__()
```

```
DXFVertex.__getitem__()
```

```
DXFVertex.__iter__()
```

```
DXFVertex.dxftags()
```

Returns all vertex components as single *DXFTag* objects.

```
DXFVertex.dxfst()
```

Returns the DXF string for all vertex components.

```
DXFVertex.clone()
```

Returns a clone of itself.

Tags

A list of DXF tags, 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.

```
class ezdxf.lldxf.Tags (list)
```

```
Tags.from_text (text)
```

Constructor from DXF string.

```
Tags.strip (tags, codes)
```

Constructor from *tags*, strips all tags with group codes in *codes* from *tags*.

Parameters *codes* – iterable of group codes

```
Tags.get_handle ()
```

Get DXF handle, raises DXFValueError if handle not exists.

```
Tags.replace_handle (new_handle)
```

Replace existing handle.

Parameters *new_handle* – new handle as hex string, e.g. ‘FFFF’

Tags.**dxftype** ()

Returns DXF type of entity, e.g. 'LINE'.

Tags.**has_tag** (*code*)

Returns True if a DXF tag with group code *code* is present else False.

Parameters *code* (*int*) – group code

Tags.**get_first_value** (*code*, *default=DXFValueError*)

Returns value of first DXF tag with given group *code* or default if default is not DXFValueError, else raises DXFValueError.

Parameters

- **code** (*int*) – group code
- **default** – default value, DXFValueError raises an exception

Tags.**get_first_tag** (*code*, *default=DXFValueError*)

Returns first DXF tag with given group *code* or default if default is not DXFValueError, else raises DXFValueError.

Parameters

- **code** (*int*) – group code
- **default** – default value, DXFValueError raises an exception

Tags.**find_all** (*code*)

Returns a list of DXF tag with given group *code*.

Parameters *code* (*int*) – group code

Tags.**tag_index** (*code*, *start=0*, *end=None*)

Return index of first DXF tag with given group code.

Parameters

- **code** (*int*) – group code
- **start** (*int*) – start index as int
- **end** (*int*) – end index as int, if None end index is length of *Tags*

Tags.**update** (*tag*)

Update first existing *tag* with same group code as *tag*, raises DXFValueError if *tag* not exists.

Parameters *tag* – new DXF tag with *code* property

Tags.**set_first** (*tag*)

Update first existing tag with same group code as *tag* or append *tag*.

Parameters *tag* – DXF tag

Tags.**remove_tags** (*codes*)

Remove all tags inplace with group codes specified in *codes*.

Parameters *codes* – iterable of group codes

Tags.**remove_tags_except** (*codes*)

Remove all tags inplace except those with group codes specified in *codes*.

Parameters *codes* – iterable of group codes

Tags.**collect_consecutive_tags** (*codes*, *start=0*, *end=None*)

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** (*int*) – start index
- **end** (*int*) – end index, if None end index is length of *Tags*

Returns collected tags as *Tags* object

ExtendedTags

Represents the extended DXF tag structure introduced with DXF R13.

```
class ezdxf.lldxf.ExtendedTags
```

TODO

Packed Tags

The following tag types store multiple tags in one object to reduce the memory footprint.

TagList

Stores multiple tags with the same group code in a standard Python list.

```
class ezdxf.lldxf.TagList
```

TODO

TagArray

Stores multiple tags with the same group code in a `array.array()` object.

```
class ezdxf.lldxf.TagArray
```

TODO

TagDict

Stores multiple key/value tags with the same group code in a standard Python dict.

```
class ezdxf.lldxf.TagDict
```

TODO

VertexArray

Stores multiple vertex tags with the same group code in a `array.array('d')` object.

```
class ezdxf.lldxf.VertexArray
```

TODO

4.4 Add-ons

4.4.1 MeshBuilder

A simple Mesh builder. 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 `render()` method, renders the mesh into a DXF *Mesh* entity. The *Mesh* entity supports ngons in AutoCAD, ngons are polygons with more than 4 vertices.

Can only create new meshes.

class `ezdxf.addons.MeshBuilder`

`MeshBuilder.add_face(vertices)`

Add a face as vertices list to the mesh. A face requires at least 3 vertices, each vertex is a (x, y, z) tuple. A face is stored as index list, which means, a face does not contain the vertex itself, but the indices of the vertices in the vertex list.

list [index v1, index v2, index v3, ...].

Parameters **vertices** – list of at least 3 vertices [(x1, y1, z1), (x2, y2, z2), (x3, y3, y3), ...]

`MeshBuilder.add_edge(vertices)`

An edge consist of two vertices [v1, v2]. Each vertex is a (x, y, z) tuple and will be added to the mesh and the resulting vertex indices will be added to the mesh edges list. The stored edge is [index v1, index v2]

Parameters **vertices** – list of 2 vertices : [(x1, y1, z1), (x2, y2, z2)]

`MeshBuilder.add_vertices(vertices)`

Add new vertices to the mesh.

e.g. adding 4 vertices to an empty mesh, returns the indices (0, 1, 2, 3), adding additional 4 vertices return s the indices (4, 5, 6, 7)

Parameters **vertices** – list of vertices, vertex as (x, y, z) tuple

Returns a tuple of vertex indices.

`MeshBuilder.add_mesh(vertices=None, faces=None, edges=None, mesh=None)`

Add another mesh to this mesh.

Parameters

- **vertices** – list of vertices, a vertex is a (x, y, z)
- **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, mesh overrides vertices, faces and edges

`MeshBuilder.transform(matrix)`

Transform actual mesh into a new mesh by applying the transformation matrix to vertices.

Parameters **matrix** – transformation matrix as *Matrix44*

Returns new *ezdxf.addons.MeshBuilder* object (same type as builder)

`MeshBuilder.translate(x=0, y=0, z=0)`

Translate mesh inplace.

`MeshBuilder.scale(sx=1, sy=1, sz=1)`

Scale mesh inplace.

`MeshBuilder.render` (*layout*, *dxfattribs=None*, *matrix=None*)
Render mesh as MESH entity into layout.

Parameters

- **layout** – ezdxf *Layout* object
- **dxfattribs** – dict of DXF attributes e.g. {'layer': 'mesh', 'color': 7}
- **matrix** – transformation matrix as *Matrix44*

`MeshBuilder.from_mesh` (*cls*, *other*)
Create new mesh from other mesh as class method.

4.4.2 MeshVertexMerger

Same functionality as *MeshBuilder*, but creates meshes with unique vertices. Resulting meshes have no doublets, but *MeshVertexMerger* needs extra memory for bookkeeping.

Can only create new meshes.

class ezdxf.addons.**MeshVertexMerger** (*MeshBuilder*)

`MeshVertexMerger.add_vertices` (*vertices*)

Add new vertices only, if no vertex with identical x, y, z coordinates already exists, else the index of the existing vertex is returned as index of the new (not added) vertex.

Parameters **vertices** – list of vertices, vertex as (x, y, z) tuple

Returns a tuple of vertex indices.

4.4.3 Forms

Basic Forms

`ezdxf.addons.circle` (*count*, *radius=1*, *elevation=0*, *close=False*)
Create polygon vertices for a circle with *radius* and *count* 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 yields *Vector* objects in counter clockwise orientation

`ezdxf.addons.ellipse` (*count*, *rx=1*, *ry=1*, *start_param=0*, *end_param=2*pi*, *elevation=0*)

Create polygon vertices for an ellipse with *rx* as x-axis radius and *ry* for y-axis radius with *count* vertices. The curve 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 yields *Vector* objects

ezdxf.addons.**euler_spiral** (*count*, *length=1*, *curvature=1*, *elevation=0*)

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.

Parameters

- **count** – count of polygon vertices
- **length** – length of curve in drawing units
- **curvature** – radius of curvature
- **elevation** – z-axis for all vertices

Returns yields *Vector* objects

ezdxf.addons.**cube** (*center=True*, *matrix=None*)

Create a cube.

Parameters

- **matrix** – transformation matrix as *Matrix44*
- **center** – ‘mass’ center of cube in (0, 0, 0) if True, else first corner at (0, 0, 0)

Returns *MeshBuilder*

ezdxf.addons.**cylinder** (*count*, *radius=1.*, *top_radius=None*, *top_center=(0, 0, 1)*, *caps=True*)

Create a cylinder.

Parameters

- **count** – profiles edge count
- **radius** – radius for bottom profile
- **top_radius** – radius for top profile, same as radius if top_radius is None
- **top_center** – location vector for the center of the top profile
- **caps** – close hull with bottom cap and top cap (as N-gons)

Returns *MeshVertexMerger*

ezdxf.addons.**cone** (*count*, *radius*, *apex=(0, 0, 1)*, *caps=True*)

Create a cone.

Parameters

- **count** – edge count of basis
- **radius** – radius of basis
- **apex** – apex of the cone
- **caps** – add a bottom face if true

Returns *MeshVertexMerger*

Form Builder

`ezdxf.addons.extrude` (*profile, path, close=True*)

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 *MeshVertexMerger*

`ezdxf.addons.from_profiles_linear` (*profiles, close=True, caps=False*)

Mesh by linear connected profiles.

Parameters

- **profiles** – list of profiles
- **close** – close profile polygon if True
- **caps** – close hull with bottom cap and top cap (as N-gons)

Returns *MeshVertexMerger*

`ezdxf.addons.from_profiles_spline` (*profiles, subdivide=4, close=True, caps=False*)

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 (as N-gons)

Returns *MeshVertexMerger*

`ezdxf.addons.rotation_form` (*count, profile, angle=2*pi, axis=(1, 0, 0)*)

Mesh 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 *MeshVertexMerger*

MengerSponge

Build a 3D Menger sponge.

class `ezdxf.addons.MengerSponge`

`MengerSponge.__init__` (*location=(0, 0, 0), length=1., level=1, kind=0*)

Parameters

- **location** – location of lower left corner as (x, y, z) tuple
- **length** – side length
- **level** – subdivide level
- **kind** – type of menger sponge:
 - 0 = original menger sponge
 - 1 = Variant XOX
 - 2 = Variant OXO
 - 3 = Jerusalem Cube

MengerSponge.**render** (*layout, merge=False, dxfattribs=None, matrix=None*)

Renders the menger sponge into layout, set *merge == 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** – ezdxf *Layout* object
- **merge** – *True* for one *Mesh* entity, *False* for individual *Mesh* entities per cube
- **dxfattribs** – dict of DXF attributes e.g. {'layer': 'mesh', 'color': 7}
- **matrix** – transformation matrix as *Matrix44*

MengerSponge.**cubes** ()

Generates all cubes of the menger sponge as individual *MeshBuilder* objects.

MengerSponge.**mesh** ()

Returns geometry as one *MeshVertexMerger* entity.

SierpinskyPyramid

Build a 3D Sierpinsky Pyramid.

class ezdxf.addons.SierpinskyPyramid

SierpinskyPyramid.__init__ (*location=(0, 0, 0), length=1., level=1, sides=4*)

Parameters

- **location** – location of base center as (x, y, z) tuple
- **length** – side length
- **level** – subdivide level
- **sides** – sides of base geometry

SierpinskyPyramid.**render** (*layout, merge=False, dxfattribs=None, matrix=None*)

Renders the sierpinsky pyramid into layout, set *merge == True* for rendering the whole sierpinsky pyramid into one *Mesh* entity, set *merge* to *False* for rendering the individual pyramids of the sierpinsky pyramid as *Mesh* entities.

Parameters

- **layout** – ezdxf *Layout* object
- **merge** – *True* for one *Mesh* entity, *False* for individual *Mesh* entities per cube

- **dxfattribs** – dict of DXF attributes e.g. {'layer': 'mesh', 'color': 7}
- **matrix** – transformation matrix as *Matrix44*

`SierpinskyPyramid.pyramids()`

Generates all pyramids of the sierpinsky pyramid as individual *MeshBuilder* objects.

`SierpinskyPyramid.mesh()`

Returns geometry as one *MeshVertexMerger* entity.

4.4.4 Spline

Render 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.

class `ezdxf.addons.Spline`

`Spline.__init__(points=None, segments=100)`

Parameters

- **points** – spline definition points
- **segments** – count of line segments for approximation, vertex count is segments+1

`Spline.render_as_fit_points(layout, degree=3, method='distance', power=.5, dxfattribs=None)`

Render a B-spline as 2d/3d polyline, where the definition points are fit points.

- 2d points in -> `add_polyline2d()`
- 3d points in -> `add_polyline3d()`

To get vertices at fit points, use `method='uniform'` and use `Spline.subdivide(count)`, where count is the sub-segment count, `count=4`, means 4 line segments between two definition points.

Parameters

- **layout** – ezdxf *Layout* object
- **degree** – degree of B-spline
- **method** – 'uniform', 'distance' or 'centripetal', calculation method for parameter t
- **power** – power for 'centripetal', default is $\text{distance}^{.5}$
- **dxfattribs** – DXF attributes for *Polyline*

`Spline.render_open_bspline(layout, degree=3, dxfattribs=None)`

Render an open uniform BSpline as 3D *Polyline*. Definition points are control points.

Parameters

- **layout** – ezdxf *Layout* object
- **degree** – degree of B-spline, (order = degree + 1)
- **dxfattribs** – DXF attributes for *Polyline*

`Spline.render_uniform_bspline(layout, degree=3, dxfattribs=None):`

Render a uniform BSpline as 3D *Polyline*. Definition points are control points.

Parameters

- **layout** – ezdxf *Layout* object
- **degree** – degree of B-spline, (order = degree + 1)

- **dxfattribs** – DXF attributes for *Polyline*

`Spline.render_closed_bspline` (*layout, degree=3, dxfattribs=None*)

Render a closed uniform BSpline as 3D *Polyline*. Definition points are control points.

Parameters

- **layout** – ezdxf *Layout* object
- **degree** – degree of B-spline, (order = degree + 1)
- **dxfattribs** – DXF attributes for *Polyline*

`Spline.render_open_rbspline` (*layout, weights, degree=3, dxfattribs=None*)

Render a rational open uniform BSpline as 3D *Polyline*. Definition points are control points.

Parameters

- **layout** – ezdxf *Layout* object
- **weights** – list of weights, requires a weight value for each defpoint.
- **degree** – degree of B-spline, (order = degree + 1)
- **dxfattribs** – DXF attributes for *Polyline*

`Spline.render_uniform_rbspline` (*layout, weights, degree=3, dxfattribs=None*)

Render a rational uniform BSpline as 3D *Polyline*. Definition points are control points.

Parameters

- **layout** – ezdxf *Layout* object
- **weights** – list of weights, requires a weight value for each defpoint.
- **degree** – degree of B-spline, (order = degree + 1)
- **dxfattribs** – DXF attributes for *Polyline*

`Spline.render_closed_rbspline` (*layout, weights, degree=3, dxfattribs=None*)

Render a rational BSpline as 3D *Polyline*. Definition points are control points.

Parameters

- **layout** – ezdxf *Layout* object
- **weights** – list of weights, requires a weight value for each defpoint.
- **degree** – degree of B-spline, (order = degree + 1)
- **dxfattribs** – DXF attributes for *Polyline*

4.4.5 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, but as with all tools, I never know if someone needs it some day.

```
class R12Spline
```

```
R12Spline.__init__(control_points, degree=2, closed=True)
```

Parameters

- **control_points** – B-spline control frame vertices as (x, y) tuples
- **degree** – degree of B-spline, 2 or 3 is valid
- **closed** – True for closed curve

`R12Spline.render` (*layout*, *segments=40*, *ucs=None*, *dxfattribs=None*)

Renders the B-spline into *layout* as 2D *Polyline* entity. Use an *UCS* to place the 2D spline in 3D space, see `R12Spline.approximate()` for more information.

Parameters

- **layout** – ezdxf *Layout*
- **segments** – count of line segments to use, vertex count is segments+1
- **ucs** – *UCS* definition, control points in ucs coordinates.
- **dxfattribs** – DXF attributes for *Polyline*

Returns the *Polyline* object

`R12Spline.approximate` (*segments=40*, *ucs=None*)

Parameters

- **segments** – count of line segments to use, vertex count is segments+1
- **ucs** – *UCS* definition, control points in ucs coordinates.

Returns list of vertices in *OCS* as *Vector* objects

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 in this UCS xy-plane, you shouldn't use z-axis coordinates, if so make sure all control points are a plane parallel to the OCS base plane (UCS xy-plane), else the result is unpredictable and depends on the used CAD application (may be crash).

4.4.6 Bezier

Render bezier curve as 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.addons.**Bezier**

`Bezier.start` (*point*, *tangent*)

Set start point and start tangent.

Parameters

- **point** – start point as (x, y, z) tuple
- **tangent** – start tangent as vector, also (x, y, z) tuple

`Bezier.append` (*point*, *tangent1*, *tangent2=None*, *segments=20*):

Append a control point with two control tangents.

Parameters

- **point** – the control point as (x, y, z) tuple
- **tangent1** – first control tangent as vector *left* of point
- **tangent2** – second control tangent as vector *right* of point, if omitted `tangent2 = -tangent1`

- **segments** – count of line segments for polyline approximation, count of line segments from previous control point to this point.

Bezier.**render** (*layout, force3d=False, dxfattribs=None*)

Render bezier curve as 2D or 3D *Polyline* entity into layout.

Parameters

- **layout** – ezdxf *Layout* object
- **force3d** – force rendering as 3D *Polyline*
- **dxfattribs** – dict of DXF attributes e.g. {'layer': 'mesh', 'color': 7}

4.4.7 EulerSpiral

Render *euler spiral* as 3D *Polyline* or *Spline*.

class ezdxf.addons.**EulerSpiral**

EulerSpiral.**__init__** (*curvature=1*)

Parameters **curvature** – Radius of curvature

EulerSpiral.**render_polyline** (*layout, length=1, segments=100, matrix=None, dxfattribs=None*)

Render euler spiral as 3D *Polyline* entity into layout.

Parameters

- **layout** – ezdxf *Layout* 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** – dict of DXF attributes e.g. {'layer': 'mesh', 'color': 7}

EulerSpiral.**render_spline** (*layout, length=1, fit_points=10, degree=3, matrix=None, dxfattribs=None*)

Render euler spiral as *Spline* entity into layout, DXF version R2000 or later required.

Parameters

- **layout** – ezdxf *Layout* object
- **length** – length measured along the spiral curve from its initial position
- **fit_points** – count of spline fit points to use
- **degree** – degree of spline
- **matrix** – transformation matrix as *Matrix44*
- **dxfattribs** – dict of DXF attributes e.g. {'layer': 'mesh', 'color': 7}

4.4.8 MText

class ezdxf.addons.**MText**

4.4.9 Table

```
class ezdxf.addons.Table
```

4.5 Howto

General preconditions:

```
import ezdxf
dwg = ezdxf.readfile("your_dxf_file.dxf")
modelspace = dwg.modelspace()
```

4.5.1 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:

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

Get attribute by tag:

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

4.5.2 Reduce Memory Footprint

- compress binary data by `Drawing.compress_binary_data()`

Warning: Data compression costs time: *memory usage vs run time*

4.5.3 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):

```
# on Windows
py -3 -m ezdxf.pp your_dxf_file.dxf
```

(continues on next page)

(continued from previous page)

```
# on Linux/Mac
python3 -m ezdxf.pp your_dxf_file.dxf
```

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.

Since ezdxf v0.8.3, a script called *dxfp* will be added to your Python script path:

```
usage: dxfp [-h] [-o] [-r] [-x] [-l] FILE [FILE ...]

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.

4.5.4 Adding New XDATA to Entity

Adding XDATA as list of tuples (group code, value):

```
dwg.appids.new('YOUR_APP_NAME') # IMPORTANT: create an APP ID entry

circle = modelspace.add_circle((10, 10), 100)
circle.tags.new_xdata('YOUR_APP_NAME',
    [
        (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.

4.5.5 A360 Viewer Problems

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

4.5.6 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.

4.6 DXF Internals

- [DXF Reference](#) provided by Autodesk.
- [DXF Developer Documentation](#) provided by Autodesk.

4.6.1 DXF File Encoding

DXF Version R2004 and prior

Drawing files of DXF versions 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

DXF	Python	Name
ANSI_874	cp874	Thai
ANSI_932	cp932	Japanese
ANSI_936	gbk	UnifiedChinese
ANSI_949	cp949	Korean
ANSI_950	cp950	TradChinese
ANSI_1250	cp1250	CentralEurope
ANSI_1251	cp1251	Cyrillic
ANSI_1252	cp1252	WesternEurope
ANSI_1253	cp1253	Greek
ANSI_1254	cp1254	Turkish
ANSI_1255	cp1255	Hebrew
ANSI_1256	cp1256	Arabic
ANSI_1257	cp1257	Baltic
ANSI_1258	cp1258	Vietnam

DXF Version R2007 and later

Starting with DXF version R2007 (AC1021) the drawing file is encoded by UTF-8, the header variable `$DWGCODE-PAGE` 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

4.6.2 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:

Group Code	Value Type
0-9	String (with the introduction of extended symbol names in DXF R2000, the 255-character limit has been increased to 256)
10-39	Double precision 3D point value
40-59	Double-precision floating-point value
40-59	Double-precision floating-point value
60-79	16-bit integer value
90-99	32-bit integer value
100	String (255-character maximum, less for Unicode strings)
102	String (255-character maximum, less for Unicode strings)
105	String representing hexadecimal (hex) handle value
110-119	Double precision floating-point value
120-129	Double precision floating-point value
130-139	Double precision floating-point value
140-149	Double precision scalar floating-point value
160-169	64-bit integer value
170-179	16-bit integer value
210-239	Double-precision floating-point value
270-279	16-bit integer value
280-289	16-bit integer value
290-299	Boolean flag value
300-309	Arbitrary text string
310-319	String representing hex value of binary chunk
320-329	String representing hex handle value
330-369	String representing hex object IDs
370-379	16-bit integer value
380-389	16-bit integer value
390-399	String representing hex handle value
400-409	16-bit integer value
410-419	String
420-429	32-bit integer value
430-439	String
440-449	32-bit integer value
450-459	Long
460-469	Double-precision floating-point value

Table 1 – continued from previous page

Group Code	Value Type
470-479	String
480-481	String representing hex handle value
999	Comment (string)
1000-1009	String (same limits as indicated with 0-9 code range)
1010-1059	Double-precision floating-point value
1060-1070	16-bit integer value
1071	32-bit integer value

Explanation for some important group codes:

Group Code	Meaning
0	DXF structure tag, entity start/end or table entries
1	The primary text value for an entity
2	A name: Attribute tag, Block name, and so on. Also used to identify a DXF section or table name.
3-4	Other textual or name values
5	Entity handle expressed as a hex string (fixed)
6	Line type name (fixed)
7	Text style name (fixed)
8	Layer name (fixed)
9	Variable name identifier (used only in HEADER section of the DXF file)
10	Primary X coordinate (start point of a Line or Text entity, center of a Circle, etc.)
11-18	Other X coordinates
20	Primary Y coordinate. 2n values always correspond to 1n values and immediately follow them in the file (expected by ezdxf!)
21-28	Other Y coordinates
30	Primary Z coordinate. 3n values always correspond to 1n and 2n values and immediately follow them in the file (expected by ezdxf!)
31-38	Other Z coordinates
39	This entity's thickness if nonzero (fixed)
40-48	Float values (text height, scale factors, etc.)
49	Repeated value - multiple 49 groups may appear in one entity for variable length tables (such as the dash lengths in the LTYPE table). A 7x group always appears before the first 49 group to specify the table length
50-58	Angles
62	Color number (fixed)
66	"Entities follow" flag (fixed), only in INSERT and POLYLINE entities
67	Identifies whether entity is in model space or paper space
68	Identifies whether viewport is on but fully off screen, is not active, or is off
69	Viewport identification number
70-78	Integer values such as repeat counts, flag bits, or modes
210, 220, 230	X, Y, and Z components of extrusion direction (fixed)
999	Comments

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 *ezdxf* 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.

Group Code	Description
1000	Strings in extended data can be up to 255 bytes long (with the 256th byte reserved for the null character)
1001	(fixed) Registered application name (ASCII string up to 31 bytes long) for XDATA
1002	(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.
1003	Name of the layer associated with the extended data
1004	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
1005	Database Handle of entities in the drawing database, see also: <i>About 1005 Group Codes</i>
1010, 1020, 1030	Three real values, in the order X, Y, Z. They can be used as a point or vector record.
1011, 1021, 1031	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.
1012, 1012, 1022	Also a 3D point that is scaled, rotated, and mirrored along with the parent (but is not moved or stretched)
1013, 1023, 1033	Also a 3D point that is scaled, rotated, and mirrored along with the parent (but is not moved or stretched)
1040	A real value
1041	Distance, a real value that is scaled along with the parent entity
1042	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
1070	A 16-bit integer (signed or unsigned)
1071	A 32-bit signed (long) integer

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.

For example, 'TESTU+7F3AU+4E4FU+89E3U+91CAU+6B63THISU+56FE'

To support the DXF unicode encoding ezdxf registers an encoding codec *dxfbackslashreplace*, defined in `ezdxf.lldxf.encoding()`.

String values can be stored with these dxf group codes:

- 0 - 9
- 100 - 101
- 300 - 309
- 410 - 419
- 430 - 439
- 470 - 479
- 999 - 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 greater 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

Tag Structure DXF R13 and later

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.

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
...
102
}
```

(102, 'YOURAPPID}') is also a valid closing tag:

```
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
```

(continues on next page)

(continued from previous page)

```
{\fArial|b0|i0|c162|p34;CHANGE;\P\P\PTTEXT}
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
0
74
0
46
0.0
```

4.6.3 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. Polyline entities are hard owners of their vertex and seqend entities. Insert entities are hard owners of their attrib and seqend entities.

When establishing a reference to another object, it is recommended that you think about whether the reference should protect an object from the PURGE command.

Arbitrary Handles

Arbitrary handles are distinct in that they are not translated to session-persistent identifiers internally, or to entity names in AutoLISP, and so on. They are stored as handles. When handle values are translated in drawing-merge operations, arbitrary handles are ignored.

In all environments, arbitrary handles can be exchanged for entity names of the current drawing by means of the handent functions. A common usage of arbitrary handles is to refer to objects in external DXF and DWG files.

About 1005 Group Codes

(1005, . . .) xdata have the same behavior and semantics as soft pointers, which means that they are translated whenever the host object is merged into a different drawing. However, 1005 items are not translated to session-persistent identifiers or internal entity names in AutoLISP and ObjectARX. They are stored as handles.

4.6.4 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). I know there exists 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 not support binary encoded DXF files (yet?).

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 (VPOR): The VPOR 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 model space and the block name *Paper_Space or *PAPER_SPACE is reserved for the *active* paper space layout. Both block definitions are empty, the content of the model space and the *active* paper space is stored in the ENTITIES section. The entities of other layouts are stored in special block definitions called *Paper_Space_{nnn}, nnn is an arbitrary but unique number.
5. **ENTITIES:** Contains all graphical entities of the model space and the *active* paper space 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
          <<< Block definition entities go here
0          <<< End BLOCKS section
ENDSEC
0          <<< Begin ENTITIES section
SECTION
2
ENTITIES
          <<< Drawing entities go here
0          <<< End ENTITIES section
ENDSEC
0          <<< End of file marker (required)
EOF

```

4.6.5 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

```

(continues on next page)

```
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

4.6.6 Coordinate Systems

AutoLISP Reference to Coordinate Systems provided by Autodesk.

To brush up you 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.algebra.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 may be was useful in the beginning 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 $\neq (0, 0, 1)$, convert OCS into WCS:

```
# circle is an DXF entity with extrusion != (0, 0, 1)
ocs = circle.ocs()
wcs_center = ocs.to_wcs(circle.dxf.center)
```

See also:

- *Object Coordinate System (OCS)* - deeper insights into OCS
- `ezdxf.algebra.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.

4.6.7 HEADER Section

Documentation to ezdxf `HeaderSection` class.

In DXF R12 an 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:

DXF Reference: [Header Variables](#)

4.6.8 CLASSES Section

The CLASSES section contains CLASS definitions which are only important for Autodesk products, some DXF entities require a class definition or AutoCAD will not open the DXF file.

The CLASSES sections was introduced with DXF AC1015 (AutoCAD Release R13).

See also:

DXF Reference: [About the DXF CLASSES Section](#)

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.

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
```

(continues on next page)

(continued from previous page)

```

1          <<< class DXF entity name; always unique
ACBBDICTIONARYWDFLT
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

```

4.6.9 TABLES Section

VIEW Table

The **VIEW** entry stores a named view of the model or a paper space 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 = model space/paper space)
0          <<< model space
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

```

(continues on next page)

(continued from previous page)

```

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
TABLE
2    <<< table name
VIEW
5    <<< handle of table, see owner tag of VIEW table entry
37C
330  <<< owner tag of table, always #0
0
100  <<< subclass marker
AcDbSymbolTable
70   <<< VIEW table (max.) count, not reliable (ignore)
9
0    <<< first VIEW table entry
VIEW
5    <<< handle
3EA
330  <<< owner, the VIEW table is the owner of the VIEW entry
37C  <<< handle of the VIEW table
100  <<< subclass marker
AcDbSymbolTableRecord
100  <<< subclass marker
AcDbViewTableRecord
2    <<< view name, from here all the same as DXF R12
VIEWNAME
70
0
    
```

(continues on next page)

(continued from previous page)

```

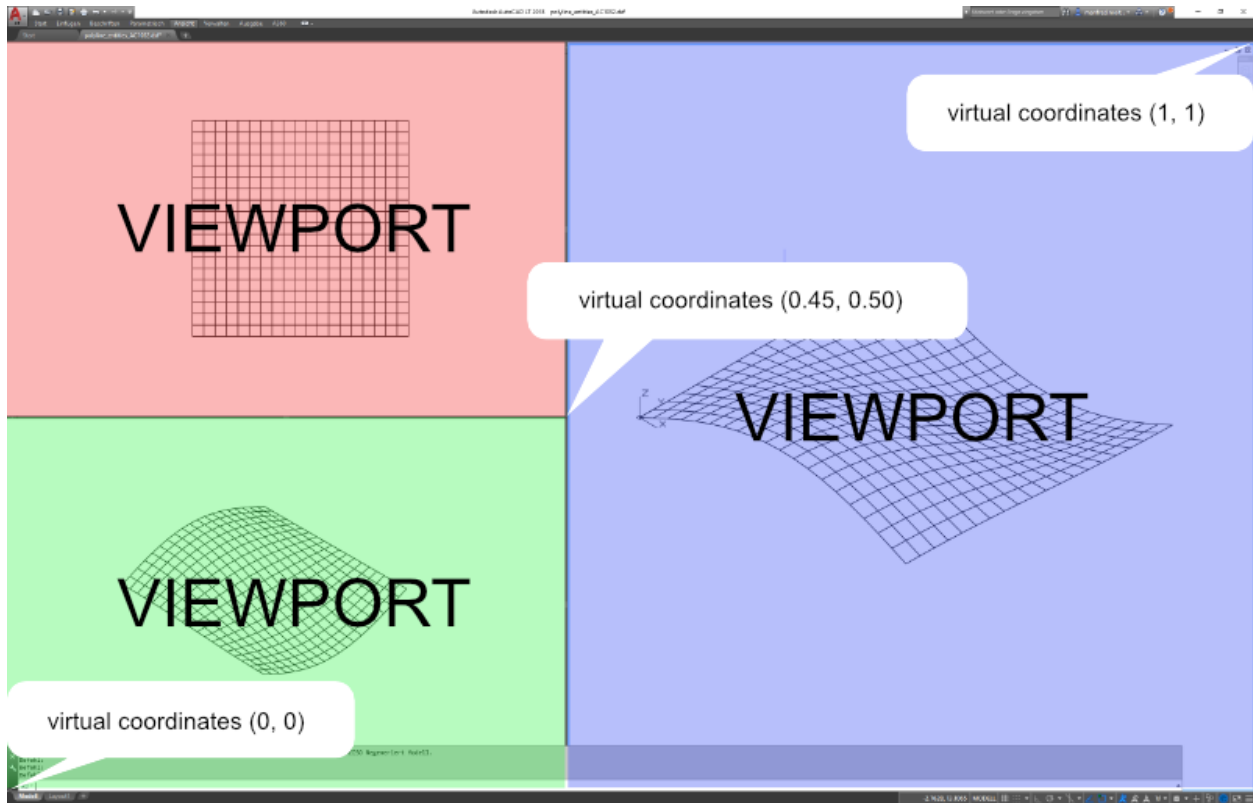
40
20.01
10
40.36
20
15.86
41
17.91
11
0.0
21
0.0
31
1.0
12
0.0
22
0.0
32
0.0
42
50.0
43
0.0
44
0.0
50
0.0
71
0
281    <<< render mode 0-6 (... too much options)
0      <<< 0= 2D optimized (classic 2D)
72    <<< UCS associated (0/1 = no/yes)
0      <<< 0 = no

```

DXF R2000+ supports additional features in the `VIEW` entry, see the [VIEW](#) table reference provided by Autodesk.

VPOR Configuration Table

The `VPOR` table stores the model space 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 DSC, the view-center-point (in DCS) defines the point in model space translated to the center point of the viewport, the view height and the aspect-ratio defines how much of the model space is displayed. AutoCAD tries to fit the model space area into the available viewport space e.g. view height is 15 units and aspect-ratio is 2.0 the model space 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 model space in the viewport. If the model space aspect-ratio is 1.0 the model space to display is 15x15 units and fits properly into the viewport area.

But tests show that the translation of the view-center-point to the middle of the viewport not always work as I expected. (still digging...)

Note: All floating point values are rounded to 2 decimal places for better readability.

DXF R12

Multi-viewport configuration with three viewports.

```

0      <<< table start
TABLE
2      <<< table type
VPORT
70     <<< VPORT table (max.) count, not reliable (ignore)
3
0      <<< first VPORT entry
VPORT
2      <<< VPORT (configuration) name
*ACTIVE
70     <<< standard flags, bit-coded
0
10     <<< lower-left corner of viewport
0.45   <<<     x value, virtual coordinates in range [0 - 1]
20     <<<     group code for y value
0.0    <<<     y value, virtual coordinates in range [0 - 1]
11     <<< upper-right corner of viewport
1.0    <<<     x value, virtual coordinates in range [0 - 1]
21     <<<     group code for y value
1.0    <<<     y value, virtual coordinates in range [0 - 1]
12     <<< view center point (in DCS), ???
13.71  <<<     x value
22     <<<     group code for y value
0.02   <<<     y value
13     <<< snap base point (in DCS)
0.0    <<<     x value
23     <<<     group code for y value
0.0    <<<     y value
14     <<< snap spacing X and Y
1.0    <<<     x value
24     <<<     group code for y value
1.0    <<<     y value
15     <<< grid spacing X and Y
0.0    <<<     x value
25     <<<     group code for y value
0.0    <<<     y value
16     <<< view direction from target point (in WCS), defines the z-axis of the DCS
1.0    <<<     x value
26     <<<     group code for y value
-1.0   <<<     y value
36     <<<     group code for z value
1.0    <<<     z value
17     <<< view target point (in WCS), defines the origin of the DCS
0.0    <<<     x value
27     <<<     group code for y value
0.0    <<<     y value
37     <<<     group code for z value
0.0    <<<     z value
40     <<< view height
35.22
41     <<< viewport aspect ratio
0.99
42     <<< lens (focal) length
50.0   <<< 50mm
43     <<< front clipping planes, offsets from target point
0.0
44     <<< back clipping planes, offsets from target point

```

(continues on next page)

(continued from previous page)

```

0.0
50    <<< snap rotation angle
0.0
51    <<< view twist angle
0.0
71    <<< view mode
0
72    <<< circle zoom percent
1000
73    <<< fast zoom setting
1
74    <<< UCSICON setting
3
75    <<< snap on/off
0
76    <<< grid on/off
0
77    <<< snap style
0
78    <<< snap isopair
0
0     <<< next VPORT entry
VPORT
2     <<< VPORT (configuration) name
*ACTIVE <<< same as first VPORT entry
70
0
10
0.0
20
0.5
11
0.45
21
1.0
12
8.21
22
9.41
...
...
0     <<< next VPORT entry
VPORT
2     <<< VPORT (configuration) name
*ACTIVE <<< same as first VPORT entry
70
0
10
0.0
20
0.0
11
0.45
21
0.5
12
2.01

```

(continues on next page)

(continued from previous page)

```

22
-9.33
...
...
0
ENDTAB

```

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     <<< VPORT table (max.) count, not reliable (ignore)
3
0      <<< first VPORT entry
VPORT
5      <<< entry handle
158B
330    <<< owner, VPORT table is owner of VPORT entry
151F
100    <<< subclass marker
AcDbSymbolTableRecord
100    <<< subclass marker
AcDbViewportTableRecord
2      <<< VPORT (configuration) name
*ACTIVE
70     <<< standard flags, bit-coded
0
10     <<< lower-left corner of viewport
0.45   <<<   x value, virtual coordinates in range [0 - 1]
20     <<<   group code for y value
0.0    <<<   y value, virtual coordinates in range [0 - 1]
11     <<< upper-right corner of viewport
1.0    <<<   x value, virtual coordinates in range [0 - 1]
21     <<<   group code for y value
1.0    <<<   y value, virtual coordinates in range [0 - 1]
12     <<< view center point (in DCS)
13.71  <<<   x value
22     <<<   group code for y value
0.38   <<<   y value
13     <<< snap base point (in DCS)
0.0    <<<   x value
23     <<<   group code for y value
0.0    <<<   y value
14     <<< snap spacing X and Y
1.0    <<<   x value

```

(continues on next page)

(continued from previous page)

```

24    <<<    group code for y value
1.0  <<<    y value
15   <<< grid spacing X and Y
0.0  <<<    x value
25   <<<    group code for y value
0.0  <<<    y value
16   <<< view direction from target point (in WCS)
1.0  <<<    x value
26   <<<    group code for y value
-1.0 <<<    y value
36   <<<    group code for z value
1.0  <<<    z value
17   <<< view target point (in WCS)
0.0  <<<    x value
27   <<<    group code for y value
0.0  <<<    y value
37   <<<    group code for z value
0.0  <<<    z value
40   <<< view height
35.22
41   <<< viewport aspect ratio
0.99
42   <<< lens (focal) length
50.0 <<< 50mm
43   <<< front clipping planes, offsets from target point
0.0
44   <<< back clipping planes, offsets from target point
0.0
50   <<< snap rotation angle
0.0
51   <<< view twist angle
0.0
71   <<< view mode
0
72   <<< circle zoom percent
1000
73   <<< fast zoom setting
1
74   <<< UCSICON setting
3
75   <<< snap on/off
0
76   <<< grid on/off
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

```

(continues on next page)

(continued from previous page)

```

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
0.45
21
1.0
12
8.21
22
9.72
...
...
0 <<< next VPORT entry
VPORT
5
158D
330
151F

```

(continues on next page)

(continued from previous page)

```

100
AcDbSymbolTableRecord
100
AcDbViewportTableRecord
2      <<< VPORT (configuration) name
*ACTIVE <<< same as first VPORT entry
70
0
10
0.0
20
0.0
11
0.45
21
0.5
12
2.01
22
-8.97
...
...
0
ENDTAB

```

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](#)

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

```

(continues on next page)

(continued from previous page)

```
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+

```
0          <<< line type table entry
LTYPE
5          <<< handle of line type
1B1
330        <<< owner handle, handle of LTYPE table
5F
100        <<< subclass marker
AcDbSymbolTableRecord
100        <<< subclass marker
AcDbLinetypeTableRecord
2          <<< line type name
CENTER
70         <<< flags
```

(continues on next page)

(continued from previous page)

```

0
3
Center  _____
72
65
73
4
40
2.0
49
1.25
74
0
49
-0.25
74
0
49
0.25
74
0
49
-0.25
74
0

```

Complex Line Type TEXT

ezdxf setup for line type 'GASLEITUNG':

```

dwg.linetypes.new('GASLEITUNG', dxfattribs={
    'description': 'Gasleitung2 ----GAS----GAS----GAS----GAS----GAS----GAS----GAS--',
    'length': 1,
    'pattern': 'A,.5,-.2,["GAS",STANDARD,S=.1,U=0.0,X=-0.1,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

```

(continues on next page)

(continued from previous page)

```
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

ezdxf setup for line type 'GRENZE2':

```
dwg.linetypes.new('GRENZE2', dxfattribs={
    'description': 'Grenze eckig ----[]-----[]-----[]-----[]----[]--',
    'length': 1.45,
    'pattern': 'A,.25,-.1,[132,ltypeshp.shx,x=-.1,s=.1],-.1,1',
})
```

SHAPE Tag Structure

```
0
LTYPE
5
615
330
5F
100 <<< subclass marker
```

(continues on next page)

```

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
1.0
74
0

```

TODO

4.6.10 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 model space and all paper space layouts, have at least a corresponding BLOCK definition in the BLOCKS section. The name of the model space 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, the *inactive* paper space layouts are named by the scheme *Paper_Spaceⁿⁿⁿⁿ, and the content of the inactive paper space layouts are stored in their BLOCK definition in the BLOCKS section.

The content entities of blocks are stored between the BLOCK and the ENDBLK entity.

BLOCKS section structure:

```

0          <<< start of a SECTION
SECTION
2          <<< start of BLOCKS section
BLOCKS
0          <<< start of 1. BLOCK definition
BLOCK
...        <<< Block content
...
0          <<< end of 1. Block definition
ENDBLK
0          <<< start of 2. BLOCK definition
BLOCK
...        <<< Block content
...
0          <<< end of 2. Block definition
ENDBLK
0          <<< end of BLOCKS section
ENDSEC
    
```

See also:

Block Management Structures Layout Management Structures

4.6.11 ENTITIES Section

TODO

4.6.12 OBJECTS Section

TODO

4.6.13 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: model space and paper space content

References are realized by simple names. The INSERT entity references the BLOCK definition by the BLOCK name, a TEXT entity defines the associated STYLE and LAYER by its name and so on, handles are not needed. Layout association of graphical entities in the ENTITIES section by the paper_space tag (67, 0 or 1), 0 or missing tag means model space, 1 means paper space. 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 paper space 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 trail and error (reverse engineering), 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 '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 also 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 paper space 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 model space and multiple paper space 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 DICTONARY objects. Starting point for this data structure is the 'root' DICTONARY with several entries to other DICTONARY objects. The root DICTONARY 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 DICTONARY:

- ACAD_COLOR: points to an empty DICTONARY
- ACAD_GROUP: supported by ezdxf
- ACAD_LAYOUT: supported by ezdxf
- ACAD_MATERIAL: points to an empty DICTONARY
- ACAD_MLEADERSTYLE: points to an empty DICTONARY
- ACAD_MLINESSTYLE: points to an empty DICTONARY
- ACAD_PLOTSETTINGS: points to an empty DICTONARY
- ACAD_PLOTSTYLENAME: points to ACBBDICTONARYWDFLT with one entry: 'Normal'
- ACAD_SCALELIST: points to an empty DICTONARY
- ACAD_TABLESTYLE: points to an empty DICTONARY
- ACAD_VISUALSTYLE: points to an empty DICTONARY

Root DICTONARY content for DXF R2018

```

0
SECTION
2      <<< start of the OBJECTS section
OBJECTS
0      <<< root DICTIONARY has to be the first object in the OBJECTS section
DICTIONARY
5      <<< handle
C
330    <<< owner tag
0      <<< always #0, has no owner
100
AcDbDictionary
281    <<< hard owner flag
1
3      <<< first entry
ACAD_CIP_PREVIOUS_PRODUCT_INFO
350    <<< handle to target (pointer)
78B    <<< points to a XRECORD with product info about the creator application
3      <<< entry with unknown meaning, if I shoul 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* paper space 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 ACBDICTIONARYWDFLT
3      <<< SCALE management
ACAD_SCALELIST

```

(continues on next page)

```

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
3      <<< VISUALSTYLE management
ACAD_VISUALSTYLE
350
506    <<< points to a DICTIONARY
3      <<< entry with unknown meaning
ACDB_RECOMPOSE_DATA
350
7F3
3      <<< entry with unknown meaning
AcDbVariableDictionary
350
7AE    <<< points to a DICTIONARY with handles to DICTIONARYVAR objects
0
DICTIONARY
...
...
0
ENDSEC

```

4.6.14 Block Management Structures

A BLOCK is a kind of layout like the model space or a paper space, 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 if 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:

```

0          <<< start of a SECTION
SECTION
2          <<< start of BLOCKS section
BLOCKS
...        <<< model space and paper space block definitions not shown,
...        <<< see layout management
...
0          <<< start of a BLOCK definition
BLOCK
8          <<< layer, what this layer definition does is another fact, I don't know_
↪ (now)
0
2          <<< block name
ArchTick
70         <<< flags
1
10         <<< base point, x
0.0
20         <<< base point, y
0.0
30         <<< base point, z
0.0
3          <<< second BLOCK name, same as (2, name)
ArchTick
1          <<< xref name, if block is an external reference
          <<< empty string!
0          <<< start of the first entity of the BLOCK
LINE
5
28E
8
0
62
0
10
500.0

```

(continues on next page)

(continued from previous page)

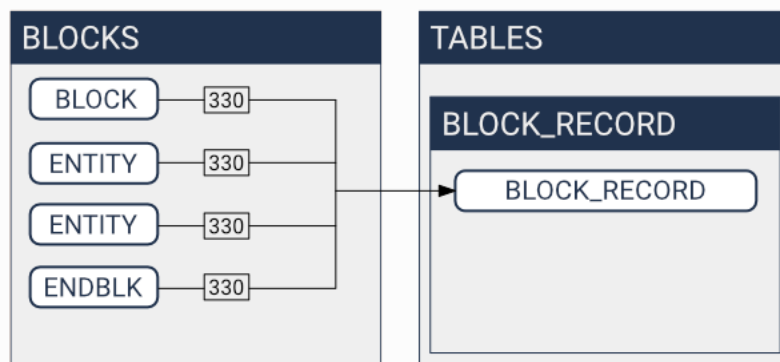
```

20
500.0
30
0.0
11
500.0
21
511.0
31
0.0
0
    <<< start of the second entity of the BLOCK
LINE
...
0.0
0
    <<< ENDBLK entity, marks the end of the BLOCK definition
ENDBLK
5
    <<< ENDBLK gets a handle by AutoCAD, but BLOCK didn't
2F2
8
    <<< as every entity, also ENDBLK requires a layer (same as BLOCK entity!)
0
0
    <<< start of next BLOCK entity
BLOCK
...
0
    <<< end BLOCK entity
ENDBLK
0
    <<< end of BLOCKS section
ENDSEC
    
```

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 model space and paper space layouts, because these layouts are also BLOCKS, with the special difference, that entities of the model space and the *active* paper space are stored in the ENTITIES section.



See also:

- *Tag Structure DXF R13 and later*
- 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
...       <<< model space and paper space block definitions not shown,
...       <<< see layout management
0          <<< start of BLOCK definition
BLOCK
5          <<< even BLOCK gets a handle now ;)
23A
330       <<< owner tag, the owner of a BLOCK is a BLOCK_RECORD in the BLOCK_RECORD_
↪table
238
100       <<< subclass marker
AcDbEntity
8         <<< layer of the BLOCK definition
0
100       <<< subclass marker
AcDbBlockBegin
2         <<< BLOCK name
ArchTick
70        <<< flags
0
10        <<< base point, x
0.0
20        <<< base point, y
0.0
30        <<< base point, z
0.0
3         <<< second BLOCK name, same as (2, name)
ArchTick
1         <<< xref name, if block is an external reference
         <<< empty string!
0         <<< start of the first entity of the BLOCK
LWPOLYLINE
5
239
330       <<< owner tag of LWPOLYLINE
238       <<< handle of the BLOCK_RECORD!
100
AcDbEntity
8
0
6
ByBlock
62
0
100
AcDbPolyline

```

(continues on next page)

(continued from previous page)

```

90
2
70
0
43
0.15
10
-0.5
20
-0.5
10
0.5
20
0.5
0 <<< ENDBLK entity, marks the end of the BLOCK definition
ENDBLK
5 <<< handle
23B
330 <<< owner tag, same BLOCK_RECORD as for the BLOCK entity
238
100 <<< subclass marker
AcDbEntity
8 <<< as every entity, also ENDBLK requires a layer (same as BLOCK entity!)
0
100 <<< subclass marker
AcDbBlockEnd
0 <<< start of the next BLOCK
BLOCK
...
0
ENDBLK
...
0 <<< end of the BLOCKS section
ENDSEC

```

DXF R13 BLOCK_RECORD structure:

```

0 <<< start of a SECTION
SECTION
2 <<< start of TABLES section
TABLES
0 <<< start of a TABLE
TABLE
2 <<< start of the BLOCK_RECORD table
BLOCK_RECORD
5 <<< handle of the table (INFO: ezdxf doesn't store tables in the entities_
↳database)
1
330 <<< owner tag of the table
0 <<< is always #0
100 <<< subclass marker
AcDbSymbolTable
70 <<< count of table entries, not reliable
4
0 <<< start of first BLOCK_RECORD entry
BLOCK_RECORD
5 <<< handle of BLOCK_RECORD, in ezdxf often referred to as 'layout key'

```

(continues on next page)

(continued from previous page)

```

1F
330      <<< owner of the BLOCK_RECORD is the BLOCK_RECORD table
1
100      <<< subclass marker
AcDbSymbolTableRecord
100      <<< subclass marker
AcDbBlockTableRecord
2        <<< name of the BLOCK or LAYOUT
*Model_Space
340      <<< pointer to the associated LAYOUT object
4AF
70       <<< AC1021 (R2007) block insertion units
0
280      <<< AC1021 (R2007) block explodability
1
281      <<< AC1021 (R2007) block scalability
0
...      <<< paper space not shown
...
0        <<< next BLOCK_RECORD
BLOCK_RECORD
5        <<< handle of BLOCK_RECORD, in ezdxf often referred to as 'layout key'
238
330      <<< owner of the BLOCK_RECORD is the BLOCK_RECORD table
1
100      <<< subclass marker
AcDbSymbolTableRecord
100      <<< subclass marker
AcDbBlockTableRecord
2        <<< name of the BLOCK
ArchTick
340      <<< pointer to the associated LAYOUT object
0        <<< #0, because BLOCK doesn't have an associated LAYOUT object
70       <<< AC1021 (R2007) block insertion units
0
280      <<< AC1021 (R2007) block explodability
1
281      <<< AC1021 (R2007) block scalability
0
0        <<< end of BLOCK_RECORD table
ENDTAB
0        <<< next TABLE
TABLE
...
0
ENDTAB
0        <<< end of TABLES section
ENDESC

```

4.6.15 Layout Management Structures

Layouts are separated entity spaces, there are three different Layout types:

1. Model space contains the 'real' world representation of the drawing subject in real world units.

2. Paper space are used to create different drawing sheets of the subject 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 model space 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 paper space layout.

DXF R13 and later supports multiple paper space layouts, the *active* layout is still called *Paper_Space, the additional *inactive* paper space layouts are named by the scheme *Paper_Space n , where the first inactive paper space is called *Paper_Space0, the second *Paper_Space1 and so on. A none consecutive numbering is tolerated by AutoCAD. The content of the inactive paper space 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 model space and paper space layouts require the DXF LAYOUT entity.

```

0
LAYOUT
5      <<< handle
59
102    <<< extension dictionary (ignore)
{ACAD_XDICTIONARY
360
1C3
102
}
102    <<< reactor (required?)
{ACAD_REACTORS
330
1A     <<< pointer to "ACAD_LAYOUT" DICTIONARY (layout management table)
102
}
330    <<< owner handle
1A     <<< pointer to "ACAD_LAYOUT" DICTIONARY (same as reactor pointer)
100    <<< PLOTSETTINGS
AcDbPlotSettings
1      <<< page setup name

2      <<< name of system printer or plot configuration file
none_device
4      <<< paper size, part in braces should follow the schema (width_x_height_unit)
↪unit is 'Inches' or 'MM'
```

(continues on next page)

(continued from previous page)

```

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_
↳1:50
1.0
143    <<< denominator of custom print scale: drawing units, 50.0 for scale 1:50
1.0
70     <<< plot layout flags, bit-coded (... too many options)
688    <<< b1010110000 = UseStandardScale(16)/PlotPlotStyle(32)/
↳PrintLineweights(128)/DrawViewportsFirst(512)
72     <<< plot paper units (0/1/2 for inches/millimeters/pixels), are pixels really_
↳supported?
0
73     <<< plot rotation (0/1/2/3 for 0deg/90deg counter-cw/upside-down/90deg cw)
1      <<< 90deg clockwise
74     <<< plot type 0-5 (... too many options)
5      <<< 5 = layout information
7      <<< current plot style name, e.g. 'acad.ctb' or 'acadlt.ctb'

75     <<< standard scale type 0-31 (... too many options)
16     <<< 16 = 1:1, also 16 if user scale type is used
147    <<< unit conversion factor
1.0    <<< for plot paper units in mm, else 0.03937... (1/25.4) for inches as plot_
↳paper units
76     <<< shade plot mode (0/1/2/3 for as displayed/wireframe/hidden/rendered)
0      <<< as displayed
77     <<< shade plot resolution level 1-5 (... too many options)
2      <<< normal
78     <<< shade plot custom DPI: 100-32767, Only applied when shade plot resolution_
↳level is set to 5 (Custom)

```

(continues on next page)

(continued from previous page)

```

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)
1.05 <<< x value
24 <<< group code for y value
0.80 <<< y value
34 <<< group code for z value
0.0 <<< z value
15 <<< maximum extents for this layout (defined by EXTMAX while this layout is_
↳current), AutoCAD default is (-1e20, -1e20, -1e20)
9.45 <<< x value
25 <<< group code for y value
7.20 <<< y value
35 <<< group code for z value
0.0 <<< z value
146 <<< elevation ???
0.0
13 <<< UCS origin (3D Point)
0.0 <<< x value
23 <<< group code for y value
0.0 <<< y value
33 <<< group code for z value

```

(continues on next page)

(continued from previous page)

```

0.0    <<<    z value
16     <<< UCS X-axis (3D vector)
1.0    <<<    x value
26     <<<    group code for y value
0.0    <<<    y value
36     <<<    group code for z value
0.0    <<<    z value
17     <<< UCS Y-axis (3D vector)
0.0    <<<    x value
27     <<<    group code for y value
1.0    <<<    y value
37     <<<    group code for z value
0.0    <<<    z value
76     <<< orthographic type of UCS 0-6 (... too many options)
0      <<< 0 = UCS is not orthographic ???
330    <<< ID/handle of required block table record
58
331    <<< ID/handle to the viewport that was last active in this layout when the_
↳layout was current
1B9
1001   <<< extended data (ignore)
...

```

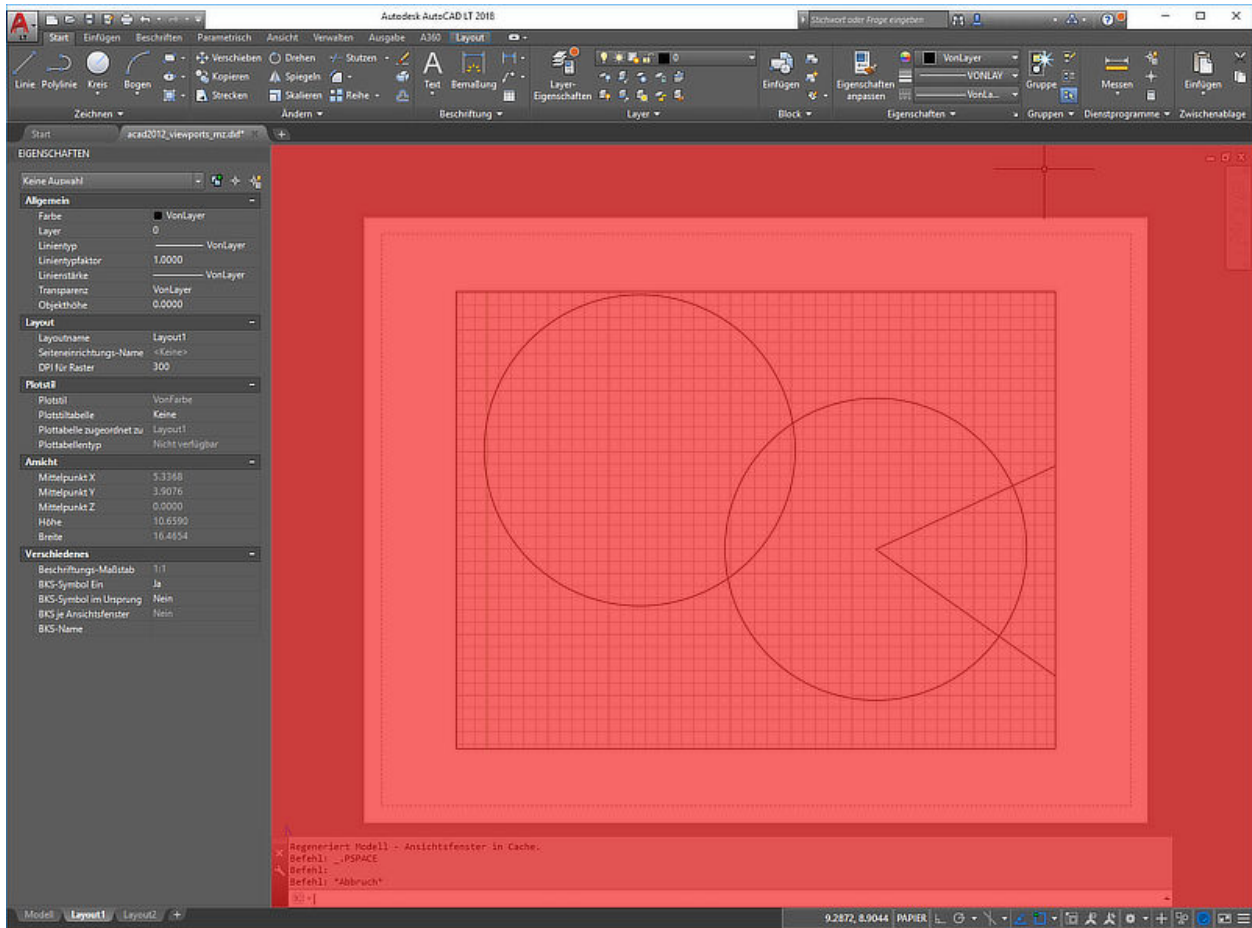
And as it seems this is also not enough for a well defined LAYOUT, at least a “main” VIEWPORT entity with ID=1 is required for paper space layouts, located in the entity space of the layout.

The model space 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* paper space layout are located in the ENTITIES section, therefore `Layout1` is not the active paper space layout.

The “main” VIEWPORT describes, how the application shows the paper space 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     <<< paper space flag
1      <<< 0 = model space; 1 = paper space
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)

(continued from previous page)

```

0.0    <<<    z value
40     <<< width in paper space units
23.55 <<< VIEW size in AutoCAD, depends on the workstation configuration
41     <<< height in paper space units
9.00  <<< VIEW size in AutoCAD, depends on the workstation configuration
68     <<< viewport status field -1/0/n
2      <<< >0 On and active. The value indicates the order of stacking for the
↳viewports, where 1 is the active viewport, 2 is the next, and so forth
69     <<< viewport ID
1      <<< "main" viewport has always ID=1
12     <<< view center point in Drawing Coordinate System (DCS), defines the center
↳point of the VIEW in relation to the LAYOUT origin
5.25  <<<    x value
22     <<<    group code for y value
4.00  <<<    y value
13     <<< snap base point in model space
0.0    <<<    x value
23     <<<    group code for y value
0.0    <<<    y value
14     <<< snap spacing in model space units
0.5    <<<    x value
24     <<<    group code for y value
0.5    <<<    y value
15     <<< grid spacing in model space units
0.5    <<<    x value
25     <<<    group code for y value
0.5    <<<    y value
16     <<< view direction vector from target (in WCS)
0.0    <<<    x value
26     <<<    group code for y value
0.0    <<<    y value
36     <<<    group code for z value
1.0    <<<    z value
17     <<< view target point
0.0    <<<    x value
27     <<<    group code for y value
0.0    <<<    y value
37     <<<    group code for z value
0.0    <<<    z value
42     <<< perspective lens length, focal length?
50.0   <<<    50mm
43     <<<    front clip plane z value
0.0    <<<    z value
44     <<<    back clip plane z value
0.0    <<<    z value
45     <<<    view height (in model space units)
9.00   <<< snap angle
50     <<< snap angle
0.0    <<< view twist angle
51     <<< view twist angle
0.0    <<< view twist angle
72     <<< circle zoom percent
1000   <<< circle zoom percent
90     <<< Viewport status bit-coded flags (... too many options)
819232 <<< b11001000000000100000
1      <<< plot style sheet name assigned to this viewport

```

(continues on next page)

(continued from previous page)

```

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
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 (... 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

```

4.6.16 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 axes 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 axes 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*
- *Solid*
- *Trace*
- *Text*
- *Attrib*
- *Attdef*
- *Shape*
- *Insert*
- *Polyline (2D)*
- *Vertex (2D)*
- *LWPolyline*
- *Hatch*
- *Image*

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 axes to go with the normal Az (the arbitrary Z axis). They will be called Ax and Ay (using *Vector*):

```
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

```
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)
```

OCS to WCS

```
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)
```

Version 0.8.9 - dev

- Release notes: <https://ezdxf.mozman.at/release-v0-8-9.html>
- IMPORTANT: Python 2 support will be dropped in ezdxf v0.9.0, because Python 2 support get more and more annoying.
- tox does not succeed with pypy 2.7, package_data config in setup.py fails in pypy 2.7
- CHANGE: refactoring of internal tag representation for smaller memory footprint, but with some speed penalty
- NEW: packed data for more memory efficient data storage for some entities
- NEW: packed data for LWPOLYLINE points, faster `__getitem__`; added `__setitem__`, `__delitem__`, `insert()` and `append()` methods; renamed `discard_points()` in `clear()`; removed `get_rstrip_points()` and `ctx manager rstrip_points()`; user defined point format;
- NEW: packed data for SPLINE, knots and weights stored as 4-byte float arrays, vertices stored as 8-byte double arrays; `Spline.get_knot_values()`, `Spline.get_weights()`, `Spline.get_control_points()` and `Spline.get_fit_points()` are deprecated, direct access to this attributes by `Spline.knot_values`, `Spline.weights`, `Spline.control_points` and `Spline.fit_points` all with a list-like interface. Knot, control point and fit point counter updated automatically, therefore counters are read only now.
- NEW: packed data for MESH, vertices, faces, edges and edge crease values stored as `array.array()`, high level interface unchanged
- NEW: `Drawing.layouts_and_blocks()`, iterate over all layouts (mode space and paper space) and all block definitions.
- NEW: `Drawing.chain_layouts_and_blocks()`, chain entity spaces of all layouts and blocks. Yields an iterator for all entities in all layouts and blocks
- NEW: `Drawing.query()`, entity query over all layouts and blocks
- NEW: `Drawing.groupby()`, groups DXF entities of all layouts and blocks by an DXF attribute or a key function
- NEW: `Layout.set_redraw_order()` and `Layout.get_redraw_order()`, to change redraw order of entities in model space and paper space layouts

- NEW: BlockLayout.is_layout_block, True if block is a model space or paper space block definition
- NEW: ezdxf.algebra.Arc helper class to create arcs from 2 points and an angle or radius, or from 3 points
- NEW: ezdxf.algebra.Arc.add_to_layout() with UCS support to create 3D arcs
- NEW: rename paper space layouts by Drawing.layouts.rename(old_name, new_name)
- NEW: Basic support for embedded objects (new in AutoCAD 2018), ezdxf reads and writes the embedded data as it is, no interpretation no modification, just enough to not break DXF files with embedded objects at saving.
- CHANGE: Drawing.blocks.delete_block(name, safe=True), new parameter save, check if block is still referenced (raises DXFValueError)
- CHANGE: Drawing.blocks.delete_all_blocks(safe=True), new parameter save, ignores blocks still referenced if safe is True
- *Basic* read support for almost all missing DXF entities/objects
 - ACAD_PROXY_GRAPHIC
 - HELIX
 - LEADER
 - LIGHT
 - MLEADER (incomplete)
 - MLINE (incomplete)
 - OLEFRAME
 - OLE2FRAME
 - SECTION
 - TABLE (incomplete)
 - TOLERANCE
 - WIPEOUT
 - ACAD_PROXY_OBJECT
 - DATATABLE
 - DICTIONARYVAR
 - DIMASSOC
 - FIELD (incomplete)
 - FIELDLIST (not documented by Autodesk)
 - IDBUFFER
 - LAYER_FILTER
 - MATERIAL
 - MLEADERSTYLE
 - MLINestyle
 - RENDER (todo)
 - SECTION (todo)
 - SORTENTSTABLE

- SPATIAL_FILTER (todo)
- SUN
- SUNSTUDY (incomplete) (no real world DXF files with SUNSTUDY for testing available)
- TABLESTYLE (incomplete)
- VBA_PROJECT (no real world DXF files with embedded VBA for testing available)
- VISUALSTYLE (todo)
- WIPEOUTVARIABLES

Version 0.8.8 - 2018-04-02

- Release notes: <https://ezdxf.mozman.at/release-v0-8-8.html>
- NEW: read/write support for GEODATA entity
- NEW: read/(limited)write support for SURFACE, EXTRUDED SURFACE, REVOLVED SURFACE, LOFTED SURFACE and SWEPT SURFACE entity
- NEW: support for extension dictionaries
- NEW: add_spline_control_frame(), create and add B-spline control frame from fit points
- NEW: add_spline_approx(), approximate B-spline by a reduced count of control points
- NEW: ezdxf.setup_linetypes(dwg), setup standard line types
- NEW: ezdxf.setup_styles(dwg), setup standard text styles
- NEW: LWPolyline.vertices() yields all points as (x, y) tuples in OCS, LWPolyline.dxf.elevation is the z-axis value
- NEW: LWPolyline.vertices_in_wcs() yields all points as (x, y, z) tuples in WCS
- NEW: basic __str__() and __repr__() support for DXF entities, returns just DXF type and handle
- NEW: bulge related function in module ezdxf.algebra.bulge
- NEW: Object Coordinate System support by DXFEntity.ocs() and OCS() class in module ezdxf.algebra
- NEW: User Coordinate System support by UCS() class in module ezdxf.algebra
- CHANGE: DXFEntity.set_app_data() and Entity.set_xdata accept also list of tuples as tags, DXFTag() is not required
- BUGFIX: entity structure validator excepts group code >= 1000 before XDATA section (used in AutoCAD Civil 3D and AutoCAD Map 3D)

Version 0.8.7 - 2018-03-04

- Release notes: <https://ezdxf.mozman.at/release-v0-8-7.html>
- NEW: entity.get_layout() returns layout in which entity resides or None if unassigned
- NEW: copy any DXF entity by entity.copy() without associated layout, add copy to any layout you want, by layout.add_entity().
- NEW: copy entity to another layout by entity.copy_to_layout(layout)
- NEW: move entity from actual layout to another layout by entity.move_to_layout(layout)
- NEW: support for splines by control points: add_open_spline(), add_closed_spline(), add_rational_spline(), add_closed_rational_spline()
- NEW: bspline_control_frame() calculates B-spline control points from fit points, but not the same as AutoCAD

- NEW: R12Spline add-on, 2d B-spline with control frame support by AutoCAD, but curve is just an approximated POLYLINE
- NEW: added `entity.get_flag_state()` and `entity.set_flag_state()` for easy access to binary coded flags
- NEW: set new \$FINGERPRINTGUID for new drawings
- NEW: set new \$VERSIONGUID on saving a drawing
- NEW: improved IMAGE support, by adding RASTERVARIABLES entity, use `Drawing.set_raster_variables(frame, quality, units)`
- BUGFIX: closing user defined image boundary path automatically, else AutoCAD crashes

Version 0.8.6 - 2018-02-17

- Release notes: <https://ezdxf.mozman.at/release-v0-8-6.html>
- NEW: ezdxf project website: <https://ezdxf.mozman.at/>
- CHANGE: create all missing tables of the TABLES sections for DXF R12
- BUGFIX: entities on new layouts will be saved
- NEW: `Layout.page_setup()` and correct 'main' viewport for DXF R2000+; For DXF R12 `page_setup()` exists, but does not provide useful results. Page setup for DXF R12 is still a mystery to me.
- NEW: `Table()`, `MText()`, `Ellipse()`, `Spline()`, `Bezier()`, `Clothoid()`, `LinearDimension()`, `RadialDimension()`, `ArcDimension()` and `AngularDimension()` composite objects from `dxfwrite` as add-ons, these add-ons support DXF R12
- NEW: geometry builder as add-ons: `MeshBuilder()`, `MeshVertexMerger()`, `MengerSponge()`, `SierpinskyPyramid()`, these add-ons require DXF R2000+ (MESH entity)
- BUGFIX: fixed invalid implementation of context manager for `r12writer`

Version 0.8.5 - 2018-01-28

- Release notes: <https://ezdxf.mozman.at/release-v0-8-5.html>
- CHANGE: block names are case insensitive 'TEST' == 'Test' (like AutoCAD)
- CHANGE: table entry (layer, linetype, style, dimstyle, ...) names are case insensitive 'TEST' == 'Test' (like AutoCAD)
- CHANGE: raises `DXFInvalidLayerName()` for invalid characters in layer names: `<>/';?*|='`
- CHANGE: audit process rewritten
- CHANGE: skip all comments, group code 999
- CHANGE: removed compression for unused sections (THUMBNAILSECTION, ACDSDATA)
- NEW: write DXF R12 files without handles: set `dwg.header['$HANDLING']=0`, default value is 1
- added subclass marker filter for R12 and prior files in `legacy_mode=True` (required for malformed DXF files)
- removed special check for Leica Disto Unit files, use `readfile(filename, legacy_mode=True)` (malformed DXF R12 file, see previous point)

Version 0.8.4 - 2018-01-14

- Release notes: <https://ezdxf.mozman.at/release-v0-8-4.html>
- NEW: Support for complex line types with text or shapes
- NEW: DXF file structure validator at SECTION level, tags outside of sections will be removed
- NEW: Basic read support for DIMENSION

- CHANGE: improved exception management, in the future ezdxf should only raise exceptions inherited from DXFError for DXF related errors, previous exception classes still work
 - DXFValueError(DXFError, ValueError)
 - DXFKeyError(DXFError, KeyError)
 - DXFAttributeError(DXFError, AttributeError)
 - DXFIndexError(DXFError, IndexError)
 - DXFTableEntryError(DXFValueError)
- speedup low level tag reader around 5%, and speedup tag compiler around 5%

Version 0.8.3 - 2018-01-02

- CHANGE: Lwpolyline - suppress yielding z coordinates if they exists (DXFStructureError: z coordinates are not defined in the DXF standard)
- NEW: setup creates a script called 'dxftp' (DXF Pretty Printer) in the Python script folder
- NEW: basic support for DXF format AC1032 introduced by AutoCAD 2018
- NEW: ezdxf use logging and writes all logs to a logger called 'ezdxf'. Logging setup is the domain of the application!
- NEW: warns about multiple block definitions with the same name in a DXF file. (DXFStructureError)
- NEW: legacy_mode parameter in ezdxf.read() and ezdxf.readfile(): tries do fix coordinate order in LINE entities (10, 11, 20, 21) by the cost of around 5% overall speed penalty at DXF file loading

Version 0.8.2 - 2017-05-01

- NEW: Insert.delete_attrib(tag) - delete ATTRIB entities from the INSERT entity
- NEW: Insert.delete_all_attribs() - delete all ATTRIB entities from the INSERT entity
- BUGFIX: setting attribs_follow=1 at INSERT entity before adding an attribute entity works

Version 0.8.1 - 2017-04-06

- NEW: added support for constant ATTRIB/ATTDEF to the INSERT (block reference) entity
- NEW: added ATTDEF management methods to BlockLayout (has_attdef, get_attdef, get_attdef_text)
- NEW: added (read/write) properties to ATTDEF/ATTRIB for setting flags (is_const, is_invisible, is_verify, is_preset)

Version 0.8.0 - 2017-03-28

- added groupby(dxfattrib='', key=None) entity query function, it is supported by all layouts and the query result container: Returns a dict, where entities are grouped by a dxfattrib or the result of a key function.
- added ezdxf.audit() for DXF error checking for drawings created by ezdxf - but not very capable yet
- dxfattribs in factory functions like add_line(dxfattribs=...), now are copied internally and stay unchanged, so they can be reused multiple times without getting modified by ezdxf.
- removed deprecated Drawing.create_layout() -> Drawing.new_layout()
- removed deprecated Layouts.create() -> Layout.new()
- removed deprecated Table.create() -> Table.new()
- removed deprecated DXFGroupTable.add() -> DXFGroupTable.new()
- BUFIX in EntityQuery.extend()

5.1 Indices and tables

- [genindex](#)
- [search](#)

e

`ezdxf.addons`, 150
`ezdxf.algebra`, 129
`ezdxf.lldxf`, 138

Symbols

- `__abs__()` (ezdxf.algebra.Vector method), 131
- `__add__()` (ezdxf.algebra.Vector method), 132
- `__bool__()` (ezdxf.algebra.Vector method), 132
- `__contains__()` (BlocksSection method), 105
- `__contains__()` (DXFGroup method), 112
- `__contains__()` (DXFGroupTable method), 113
- `__contains__()` (Layout method), 58
- `__contains__()` (Layouts method), 56
- `__contains__()` (Table method), 45
- `__copy__()` (ezdxf.algebra.Matrix44 method), 133
- `__copy__()` (ezdxf.algebra.Vector method), 131
- `__deepcopy__()` (ezdxf.algebra.Vector method), 131
- `__delitem__()` (LWPPolyline method), 81
- `__div__()` (ezdxf.algebra.Vector method), 132
- `__eq__()` (ezdxf.algebra.Vector method), 132
- `__getitem__()` (BlocksSection method), 105
- `__getitem__()` (Face method), 75
- `__getitem__()` (HeaderSection method), 43
- `__getitem__()` (LWPPolyline method), 80
- `__getitem__()` (MeshVertexCache method), 74
- `__getitem__()` (Polyline method), 72
- `__getitem__()` (ezdxf.algebra.Matrix44 method), 134
- `__getitem__()` (ezdxf.algebra.Vector method), 131
- `__getitem__()` (ezdxf.lldxf.DXFTag method), 138
- `__getitem__()` (ezdxf.lldxf.DXFVertex method), 139
- `__hash__()` (ezdxf.algebra.Vector method), 131
- `__iadd__()` (MTextData method), 83
- `__imul__()` (ezdxf.algebra.Matrix44 method), 134
- `__init__()` (EntityQuery method), 120
- `__init__()` (Importer method), 117
- `__init__()` (R12FastStreamWriter method), 123
- `__init__()` (ezdxf.addons.EulerSpiral method), 150
- `__init__()` (ezdxf.addons.MengerSponge method), 145
- `__init__()` (ezdxf.addons.R12Spline method), 148
- `__init__()` (ezdxf.addons.SierpinskyPyramid method), 146
- `__init__()` (ezdxf.addons.Spline method), 147
- `__init__()` (ezdxf.algebra.OCS method), 127
- `__init__()` (ezdxf.algebra.UCS method), 127
- `__iter__()` (BlocksSection method), 105
- `__iter__()` (CustomVars method), 44
- `__iter__()` (DXFGroup method), 112
- `__iter__()` (DXFGroupTable method), 113
- `__iter__()` (Face method), 75
- `__iter__()` (Layout method), 58
- `__iter__()` (Layouts method), 56
- `__iter__()` (Table method), 45
- `__iter__()` (ezdxf.algebra.Matrix44 method), 134
- `__iter__()` (ezdxf.algebra.Vector method), 131
- `__iter__()` (ezdxf.lldxf.DXFTag method), 138
- `__iter__()` (ezdxf.lldxf.DXFVertex method), 139
- `__len__()` (CustomVars method), 44
- `__len__()` (DXFGroup method), 112
- `__len__()` (DXFGroupTable method), 113
- `__len__()` (Face method), 75
- `__len__()` (LWPPolyline method), 80
- `__len__()` (Layouts method), 56
- `__len__()` (Polyline method), 72
- `__len__()` (Table method), 45
- `__len__()` (ezdxf.algebra.Vector method), 131
- `__lt__()` (ezdxf.algebra.Vector method), 132
- `__mul__()` (ezdxf.algebra.Matrix44 method), 134
- `__mul__()` (ezdxf.algebra.Vector method), 132
- `__neg__()` (ezdxf.algebra.Vector method), 132
- `__radd__()` (ezdxf.algebra.Vector method), 132
- `__rdiv__()` (ezdxf.algebra.Vector method), 132
- `__repr__()` (ezdxf.algebra.Matrix44 method), 133
- `__repr__()` (ezdxf.algebra.Vector method), 131
- `__repr__()` (ezdxf.lldxf.DXFTag method), 138
- `__repr__()` (ezdxf.lldxf.DXFVertex method), 139
- `__rmul__()` (ezdxf.algebra.Vector method), 132
- `__rsub__()` (ezdxf.algebra.Vector method), 132
- `__rtruediv__()` (ezdxf.algebra.Vector method), 132
- `__setitem__()` (HeaderSection method), 43
- `__setitem__()` (LWPPolyline method), 80
- `__setitem__()` (MeshVertexCache method), 74
- `__setitem__()` (ezdxf.algebra.Matrix44 method), 134
- `__str__()` (ModelerGeometryData method), 98

__str__() (ezdxf.algebra.Vector method), 131
 __str__() (ezdxf.lldxf.DXFTag method), 138
 __str__() (ezdxf.lldxf.DXFVertex method), 139
 __sub__() (ezdxf.algebra.Vector method), 132
 __truediv__() (ezdxf.algebra.Vector method), 132
 3DFace (built-in class), 76
 3DSolid (built-in class), 99

A

acad_version (Drawing attribute), 39
 add_3dface() (Layout method), 58
 add_3dface() (R12FastStreamWriter method), 124
 add_3dsolid() (Layout method), 61
 add_arc() (EdgePath method), 88
 add_arc() (Layout method), 58
 add_arc() (R12FastStreamWriter method), 123
 add_attdef() (BlockLayout method), 62
 add_attr() (Insert method), 108
 add_attr() (Layout method), 59
 add_auto_blockref() (Layout method), 59
 add_blockref() (Layout method), 58
 add_body() (Layout method), 61
 add_circle() (Layout method), 58
 add_circle() (R12FastStreamWriter method), 123
 add_closed_rational_spline() (Layout method), 60
 add_closed_spline() (Layout method), 59
 add_edge() (ezdxf.addons.MeshBuilder method), 142
 add_edge() (MeshData method), 92
 add_edge_path() (BoundaryPathData method), 87
 add_ellipse() (EdgePath method), 88
 add_ellipse() (Layout method), 58
 add_entity() (Layout method), 61
 add_face() (ezdxf.addons.MeshBuilder method), 142
 add_face() (MeshData method), 92
 add_hatch() (Layout method), 61
 add_image() (Layout method), 61
 add_image_def() (Drawing method), 41
 add_line() (EdgePath method), 88
 add_line() (Layout method), 58
 add_line() (PatternData method), 91
 add_line() (R12FastStreamWriter method), 123
 add_lwpolyline() (Layout method), 59
 add_mesh() (ezdxf.addons.MeshBuilder method), 142
 add_mtext() (Layout method), 59
 add_open_spline() (Layout method), 59
 add_point() (Layout method), 58
 add_point() (R12FastStreamWriter method), 124
 add_polyface() (Layout method), 59
 add_polyline() (R12FastStreamWriter method), 124
 add_polyline2d() (Layout method), 59
 add_polyline3d() (Layout method), 59
 add_polyline_path() (BoundaryPathData method), 87
 add_polymesh() (Layout method), 59
 add_rational_spline() (Layout method), 59

add_ray() (Layout method), 59
 add_region() (Layout method), 61
 add_shape() (Layout method), 59
 add_solid() (Layout method), 58
 add_solid() (R12FastStreamWriter method), 124
 add_spline() (EdgePath method), 89
 add_spline() (Layout method), 59
 add_spline_approx() (Layout method), 60
 add_spline_control_frame() (Layout method), 60
 add_text() (Layout method), 58
 add_text() (R12FastStreamWriter method), 125
 add_to_layout() (ezdxf.algebra.Arc method), 138
 add_trace() (Layout method), 58
 add_underlay() (Layout method), 61
 add_underlay_def() (Drawing method), 42
 add_vertices() (ezdxf.addons.MeshBuilder method), 142
 add_vertices() (ezdxf.addons.MeshVertexMerger method), 143
 add_xline() (Layout method), 59
 add_xref_def() (Drawing method), 42
 adjust_for_background (Underlay attribute), 105
 align_angle (ExtrudedSurface.dxf attribute), 100
 align_angle (SweptSurface.dxf attribute), 103
 align_direction (LoftedSurface.dxf attribute), 101
 align_point (AttDef.dxf attribute), 109
 align_point (Attrib.dxf attribute), 110
 align_point (Text.dxf attribute), 68
 align_start (ExtrudedSurface.dxf attribute), 100
 align_start (SweptSurface.dxf attribute), 103
 angle (PatternDefinitionLine attribute), 91
 angle_between() (ezdxf.algebra.Vector method), 132
 angle_deg (ezdxf.algebra.Vector attribute), 130
 angle_rad (ezdxf.algebra.Vector attribute), 130
 append() (CustomVars method), 44
 append() (LWPolyline method), 80
 append() (MTextData method), 83
 append_face() (Polyface method), 74
 append_faces() (Polyface method), 74
 append_points() (LWPolyline method), 80
 append_vertices() (Polyline method), 72
 AppID (built-in class), 54
 appids (Drawing attribute), 41
 approximate() (ezdxf.addons.R12Spline method), 149
 approximate() (ezdxf.algebra.BSpline method), 135
 Arc (built-in class), 67
 Arc (class in ezdxf.algebra), 136
 arc_length_parameterization (LoftedSurface.dxf attribute), 101
 arc_to_bulge() (in module ezdxf.algebra), 126
 ArcEdge (built-in class), 89
 aspect_ratio (VPort.dxf attribute), 53
 associative (Hatch.dxf attribute), 84
 attachment_point (MText.dxf attribute), 81
 Attdef (built-in class), 108

attdefs() (BlockLayout method), 62
 Attrib (built-in class), 110
 attribs() (Insert method), 108
 axis_point (RevolvedSurface.dxf attribute), 102
 axis_rotate() (ezdxf.algebra.Matrix44 method), 133
 axis_vector (RevolvedSurface.dxf attribute), 102

B

back_clipping (View.dxf attribute), 54
 back_clipping (VPort.dxf attribute), 53
 bank (ExtrudedSurface.dxf attribute), 100
 bank (SweptSurface.dxf attribute), 103
 base_point (Block.dxf attribute), 106
 base_point (PatternDefinitionLine attribute), 91
 base_point_set (ExtrudedSurface.dxf attribute), 100
 base_point_set (SweptSurface.dxf attribute), 103
 basis_values() (ezdxf.algebra.BSpline method), 135
 Bezier (class in ezdxf.addons), 149
 bgcolor (Hatch attribute), 85
 bigfont (Style.dxf attribute), 48
 blend_crease (Mesh.dxf attribute), 92
 block (BlockLayout attribute), 62
 Block (built-in class), 106
 block_record (GeoData.dxf attribute), 116
 BlockLayout (built-in class), 62
 BlockRecord (built-in class), 55
 blocks (Drawing attribute), 40
 BlocksSection (built-in class), 105
 Body (built-in class), 98
 BoundaryPathData (built-in class), 87
 brightness (Image.dxf attribute), 96
 BSpline (class in ezdxf.algebra), 135, 136
 bspline_control_frame() (in module ezdxf.algebra), 125
 BSplineClosed (class in ezdxf.algebra), 136
 bulge (Vertex.dxf attribute), 72
 bulge_3_points() (in module ezdxf.algebra), 127
 bulge_center() (in module ezdxf.algebra), 126
 bulge_radius() (in module ezdxf.algebra), 126
 bulge_to_arc() (in module ezdxf.algebra), 126

C

center (Arc.dxf attribute), 67
 center (ArcEdge attribute), 89
 center (Circle.dxf attribute), 67
 center (Ellipse.dxf attribute), 77
 center (ezdxf.algebra.Arc attribute), 136
 center_point (View.dxf attribute), 54
 center_point (VPort.dxf attribute), 53
 centered (GradientData attribute), 91
 chain() (ezdxf.algebra.Matrix44 method), 134
 chain_layouts_and_blocks() (Drawing method), 41
 char_height (MText.dxf attribute), 81
 check_entity_tag_structures (ezdxf.options attribute), 39
 Circle (built-in class), 66

circle() (in module ezdxf.addons), 143
 circle_zoom (VPort.dxf attribute), 53
 class_id (ExtrudedSurface.dxf attribute), 100
 class_id (RevolvedSurface.dxf attribute), 102
 cleanup() (Drawing method), 42
 cleanup() (DXFGroupTable method), 113
 clear() (BoundaryPathData method), 87
 clear() (CustomVars method), 44
 clear() (DXFGroup method), 113
 clear() (DXFGroupTable method), 113
 clear() (EdgePath method), 88
 clear() (LWPolyline method), 80
 clear() (PatternData method), 91
 clear() (PolylinePath method), 88
 clip_mode (Image.dxf attribute), 97
 clipping (Image.dxf attribute), 96
 clipping (Underlay attribute), 105
 clipping_boundary_type (Image.dxf attribute), 96
 clone() (ezdxf.lldxf.DXFTag method), 138
 clone() (ezdxf.lldxf.DXFVertex method), 139
 close() (Polyline method), 71
 close() (R12FastStreamWriter method), 123
 close_to_axis (RevolvedSurface.dxf attribute), 102
 closed (LWPolyline attribute), 79
 closed (Spline attribute), 94
 closed_surfaces (LoftedSurface.dxf attribute), 101
 code (ezdxf.lldxf.DXFTag attribute), 138
 code (ezdxf.lldxf.DXFVertex attribute), 139
 collect_consecutive_tags() (ezdxf.lldxf.Tags method), 140
 color (GraphicEntity.dxf attribute), 64, 65
 color (Layer.dxf attribute), 46
 color1 (GradientData attribute), 91
 color2 (GradientData attribute), 91
 color_name (GraphicEntity.dxf attribute), 65
 column_count (Insert.dxf attribute), 108
 column_spacing (Insert.dxf attribute), 108
 columns() (ezdxf.algebra.Matrix44 method), 135
 compress_binary_data (ezdxf.options attribute), 39
 compress_binary_data() (Drawing method), 42
 cone() (in module ezdxf.addons), 144
 const_width (LWPolyline.dxf attribute), 79
 contrast (Image.dxf attribute), 96
 contrast (underlay.dxf attribute), 104
 control_point_tolerance (Spline.dxf attribute), 94
 control_points (ezdxf.algebra.BSpline attribute), 135
 control_points (Spline attribute), 94
 control_points (SplineData attribute), 95
 control_points (SplineEdge attribute), 90
 ControlPoints (built-in class), 95
 coordinate_projection_radius (GeoData.dxf attribute), 116
 coordinate_type (GeoData.dxf attribute), 116
 copy() (DXFObject method), 114

copy() (ezdxf.algebra.Matrix44 method), 133
 copy() (ezdxf.algebra.Vector method), 131
 copy() (GraphicEntity method), 63
 copy_to_layout() (GraphicEntity method), 63
 count (ezdxf.algebra.BSpline attribute), 135
 count (LWPPolyline.dxf attribute), 79
 count_boundary_points (Image.dxf attribute), 97
 cross() (ezdxf.algebra.Vector method), 132
 cube() (in module ezdxf.addons), 144
 cubes() (ezdxf.addons.MengerSponge method), 146
 custom_vars (HeaderSection attribute), 43
 CustomVars (built-in class), 43
 cylinder() (in module ezdxf.addons), 144

D

dash_length_items (PatternDefinitionLine attribute), 91
 DBSpline (class in ezdxf.algebra), 136
 DBSplineClosed (class in ezdxf.algebra), 136
 DBSplineU (class in ezdxf.algebra), 136
 default_end_width (Polyline.dxf attribute), 70
 default_start_width (Polyline.dxf attribute), 70
 degree (ezdxf.algebra.BSpline attribute), 135
 degree (Spline.dxf attribute), 93
 degree (SplineEdge attribute), 90
 del_dxf_attrib() (DXFObject method), 114
 del_dxf_attrib() (GraphicEntity method), 64
 delete() (DXFGroupTable method), 113
 delete() (Layouts method), 56
 delete_all_attribs() (Insert method), 108
 delete_all_blocks() (BlockSection method), 106
 delete_all_entities() (Layout method), 62
 delete_attrib() (Insert method), 108
 delete_block() (BlockSection method), 106
 delete_entity() (Layout method), 62
 delete_layout() (Drawing method), 41
 delete_vertices() (Polyline method), 72
 description (Linetype.dxf attribute), 48
 design_point (GeoData.dxf attribute), 116
 determinant() (ezdxf.algebra.Matrix44 method), 135
 dimadec (DimStyle.dxf attribute), 51
 dimalt (DimStyle.dxf attribute), 50
 dimaltd (DimStyle.dxf attribute), 51
 dimaltf (DimStyle.dxf attribute), 50
 dimaltrnd (DimStyle.dxf attribute), 50
 dimaltrtd (DimStyle.dxf attribute), 51
 dimalttz (DimStyle.dxf attribute), 52
 dimaltu (DimStyle.dxf attribute), 51
 dimaltz (DimStyle.dxf attribute), 52
 dimapost (DimStyle.dxf attribute), 49
 dimasz (DimStyle.dxf attribute), 49
 dimatfit (DimStyle.dxf attribute), 52
 dimaunit (DimStyle.dxf attribute), 51
 dimazin (DimStyle.dxf attribute), 50
 dimblk (DimStyle.dxf attribute), 49

dimblk1 (DimStyle.dxf attribute), 49
 dimblk1_handle (DimStyle.dxf attribute), 52
 dimblk2 (DimStyle.dxf attribute), 49
 dimblk2_handle (DimStyle.dxf attribute), 52
 dimblk_handle (DimStyle.dxf attribute), 52
 dimcen (DimStyle.dxf attribute), 50
 dimclrd (DimStyle.dxf attribute), 51
 dimclre (DimStyle.dxf attribute), 51
 dimclrt (DimStyle.dxf attribute), 51
 dimdec (DimStyle.dxf attribute), 51
 dimdle (DimStyle.dxf attribute), 49
 dimdli (DimStyle.dxf attribute), 49
 dimdsep (DimStyle.dxf attribute), 51
 dimexe (DimStyle.dxf attribute), 49
 dimexo (DimStyle.dxf attribute), 49
 dimfit (DimStyle.dxf attribute), 52
 dimfrac (DimStyle.dxf attribute), 51
 dimgap (DimStyle.dxf attribute), 50
 dimjust (DimStyle.dxf attribute), 52
 dimldrblk_handle (DimStyle.dxf attribute), 52
 dimlfac (DimStyle.dxf attribute), 50
 dimlim (DimStyle.dxf attribute), 50
 dimlunit (DimStyle.dxf attribute), 51
 dimlwd (DimStyle.dxf attribute), 52
 dimlwe (DimStyle.dxf attribute), 52
 dimpost (DimStyle.dxf attribute), 49
 dimrnd (DimStyle.dxf attribute), 49
 dimsah (DimStyle.dxf attribute), 51
 dimscale (DimStyle.dxf attribute), 49
 dimsd1 (DimStyle.dxf attribute), 52
 dimsd2 (DimStyle.dxf attribute), 52
 dimse1 (DimStyle.dxf attribute), 50
 dimse2 (DimStyle.dxf attribute), 50
 dimsoxd (DimStyle.dxf attribute), 51
 DimStyle (built-in class), 48
 dimstyles (Drawing attribute), 40
 dimtad (DimStyle.dxf attribute), 50
 dimtdec (DimStyle.dxf attribute), 51
 dimtfac (DimStyle.dxf attribute), 50
 dimtih (DimStyle.dxf attribute), 50
 dimtix (DimStyle.dxf attribute), 51
 dimtm (DimStyle.dxf attribute), 49
 dimtmove (DimStyle.dxf attribute), 52
 dimtofl (DimStyle.dxf attribute), 51
 dimtoh (DimStyle.dxf attribute), 50
 dimtol (DimStyle.dxf attribute), 50
 dimtolj (DimStyle.dxf attribute), 52
 dimtp (DimStyle.dxf attribute), 49
 dimtsz (DimStyle.dxf attribute), 50
 dimtvp (DimStyle.dxf attribute), 50
 dimtxsty_handle (DimStyle.dxf attribute), 52
 dimtxt (DimStyle.dxf attribute), 49
 dimunit (DimStyle.dxf attribute), 51
 dimupt (DimStyle.dxf attribute), 52

dimzin (DimStyle.dxf attribute), 50
 direction_point (View.dxf attribute), 54
 direction_point (VPort.dxf attribute), 53
 distance() (ezdxf.algebra.Vector method), 132
 dot() (ezdxf.algebra.Vector method), 132
 draft_angle (ExtrudedSurface.dxf attribute), 100
 draft_angle (RevolvedSurface.dxf attribute), 102
 draft_angle (SweptSurface.dxf attribute), 103
 draft_end_distance (ExtrudedSurface.dxf attribute), 100
 draft_end_distance (SweptSurface.dxf attribute), 103
 draft_start_distance (ExtrudedSurface.dxf attribute), 100
 draft_start_distance (SweptSurface.dxf attribute), 103
 Drawing (built-in class), 39
 drawing (DXFObject attribute), 113
 drawing (GraphicEntity attribute), 63
 dxf (DXFObject attribute), 113
 dxf (GraphicEntity attribute), 63
 dxf_attrib_exists() (DXFObject method), 114
 dxf_attrib_exists() (GraphicEntity method), 64
 dxfattribs() (DXFObject method), 114
 dxfattribs() (GraphicEntity method), 64
 DXFBinaryTag (class in ezdxf.lldxf), 139
 dxffactory (Drawing attribute), 40
 dxffactory (DXFObject attribute), 113
 dxffactory (GraphicEntity attribute), 63
 DXFGroup (built-in class), 112
 DXFGroupTable (built-in class), 113
 DXFObject (built-in class), 113
 dxfstr() (ezdxf.lldxf.DXFTag method), 138
 dxfstr() (ezdxf.lldxf.DXFVertex method), 139
 DXFTag (class in ezdxf.lldxf), 138
 dxftags() (ezdxf.lldxf.DXFVertex method), 139
 dxftype() (DXFObject method), 113
 dxftype() (ezdxf.lldxf.Tags method), 140
 dxftype() (GraphicEntity method), 63
 dxfversion (Drawing attribute), 39
 DXFVertex (class in ezdxf.lldxf), 139

E

edge_crease_values (MeshData attribute), 92
 EdgePath (built-in class), 88
 edges (EdgePath attribute), 88
 edges (MeshData attribute), 92
 edit_boundary() (Hatch method), 85
 edit_data() (3DSolid method), 99
 edit_data() (Body method), 98
 edit_data() (DXFGroup method), 112
 edit_data() (Mesh method), 92
 edit_data() (MText method), 82
 edit_data() (Region method), 99
 edit_data() (Spline method), 95
 edit_data() (Surface method), 100
 edit_gradient() (Hatch method), 86
 edit_pattern() (Hatch method), 85

elevation (Hatch.dxf attribute), 84
 elevation (LWPolyline.dxf attribute), 79
 elevation (Polyline.dxf attribute), 70
 Ellipse (built-in class), 77
 ellipse() (in module ezdxf.addons), 143
 EllipseEdge (built-in class), 90
 encoding (Drawing attribute), 39
 end (Line.dxf attribute), 66
 end (LineEdge attribute), 89
 end_angle (Arc.dxf attribute), 67
 end_angle (ArcEdge attribute), 89
 end_angle (EllipseEdge attribute), 90
 end_angle (ezdxf.algebra.Arc attribute), 137
 end_angle_rad (ezdxf.algebra.Arc attribute), 137
 end_draft_angle (LoftedSurface.dxf attribute), 101
 end_draft_distance (RevolvedSurface.dxf attribute), 102
 end_draft_magnitude (LoftedSurface.dxf attribute), 101
 end_param (Ellipse.dxf attribute), 78
 end_tangent (Spline.dxf attribute), 94
 end_tangent (SplineEdge attribute), 90
 end_width (Vertex.dxf attribute), 72
 entities (Drawing attribute), 40
 EntityQuery (built-in class), 120
 euler_spiral() (in module ezdxf.addons), 144
 EulerSpiral (class in ezdxf.addons), 150
 extend() (DXFGroup method), 112
 extend() (EntityQuery method), 120
 ExtendedTags (class in ezdxf.lldxf), 141
 extrude() (in module ezdxf.addons), 145
 ExtrudedSurface (built-in class), 100
 extrusion (GraphicEntity.dxf attribute), 64, 65
 extrusion (underlay.dxf attribute), 104
 ezdxf.addons (module), 142, 143, 147, 150
 ezdxf.algebra (module), 125, 127, 129, 132, 135, 136
 ezdxf.lldxf (module), 138
 ezdxf.new() (built-in function), 38
 ezdxf.read() (built-in function), 38
 ezdxf.readfile() (built-in function), 38

F

Face (built-in class), 75
 face_record (Face attribute), 75
 faces (MeshData attribute), 92
 faces() (Polyface method), 74
 fade (Image.dxf attribute), 96
 fade (underlay.dxf attribute), 104
 fast_mul() (ezdxf.algebra.Matrix44 method), 134
 fast_zoom (VPort.dxf attribute), 53
 field_length (AttDef.dxf attribute), 109
 filename (Drawing attribute), 40
 filename (ImageDef.dxf attribute), 115
 filename (UnderlayDefinition.dxf attribute), 115
 find_all() (ezdxf.lldxf.Tags method), 140
 find_shx() (StyleTable method), 45

- fit_points (Spline attribute), 94
 - fit_points (SplineData attribute), 95
 - fit_points (SplineEdge attribute), 90
 - fit_tolerance (Spline.dxf attribute), 94
 - FitPoints (built-in class), 95
 - flags (AppID.dxf attribute), 55
 - flags (Block.dxf attribute), 106
 - flags (DimStyle.dxf attribute), 49
 - flags (Image.dxf attribute), 96
 - flags (Layer.dxf attribute), 46
 - flags (LWPPolyline.dxf attribute), 79
 - flags (Polyline.dxf attribute), 70
 - flags (Spline.dxf attribute), 93
 - flags (Style.dxf attribute), 47
 - flags (UCS.dxf attribute), 55
 - flags (underlay.dxf attribute), 104
 - flags (Vertex.dxf attribute), 73
 - flags (View.dxf attribute), 54
 - flags (VPort.dxf attribute), 53
 - flow_direction (MText.dxf attribute), 81
 - font (Style.dxf attribute), 48
 - freeze() (Layer method), 46
 - from_2p_angle() (ezdxf.algebra.Arc method), 137
 - from_2p_radius() (ezdxf.algebra.Arc method), 137
 - from_3p() (ezdxf.algebra.Arc method), 137
 - from_deg_angle() (ezdxf.algebra.Vector method), 131
 - from_mesh() (ezdxf.addons.MeshBuilder method), 143
 - from_profiles_linear() (in module ezdxf.addons), 145
 - from_profiles_spline() (in module ezdxf.addons), 145
 - from_rad_angle() (ezdxf.algebra.Vector method), 131
 - from_text() (ezdxf.lldxf.Tags method), 139
 - from_wcs() (ezdxf.algebra.OCS method), 127
 - from_wcs() (ezdxf.algebra.UCS method), 128
 - from_x_axis_and_point_in_xy() (ezdxf.algebra.UCS method), 128
 - from_x_axis_and_point_in_xz() (ezdxf.algebra.UCS method), 128
 - from_y_axis_and_point_in_xy() (ezdxf.algebra.UCS method), 128
 - from_y_axis_and_point_in_yz() (ezdxf.algebra.UCS method), 128
 - from_z_axis_and_point_in_xz() (ezdxf.algebra.UCS method), 129
 - from_z_axis_and_point_in_yz() (ezdxf.algebra.UCS method), 129
 - front_clipping (View.dxf attribute), 54
 - front_clipping (VPort.dxf attribute), 53
- ## G
- generate() (ezdxf.algebra.Vector method), 131
 - geo_rss_tag (GeoData.dxf attribute), 116
 - GeoData (built-in class), 115
 - get() (BlocksSection method), 105
 - get() (CustomVars method), 44
 - get() (DXFGroupTable method), 113
 - get() (Layouts method), 56
 - get() (Table method), 44
 - get_acis_data() (3DSolid method), 99
 - get_acis_data() (Body method), 98
 - get_acis_data() (Region method), 98
 - get_acis_data() (Surface method), 100
 - get_align() (Attdef method), 110
 - get_align() (Attrib method), 111
 - get_align() (Text method), 69
 - get_attdef() (BlockLayout method), 62
 - get_attdef_text() (BlockLayout method), 63
 - get_attrib() (Insert method), 108
 - get_attrib_text() (Insert method), 108
 - get_boundary() (Image method), 97
 - get_boundary() (Underlay method), 105
 - get_col() (ezdxf.algebra.Matrix44 method), 133
 - get_color() (Layer method), 47
 - get_config() (ViewportTable method), 45
 - get_coordinate_system_definition() (GeoData method), 117
 - get_dxf_attrib() (DXFObject method), 114
 - get_dxf_attrib() (GraphicEntity method), 63
 - get_dxf_entity() (Drawing method), 43
 - get_first_tag() (ezdxf.lldxf.Tags method), 140
 - get_first_value() (ezdxf.lldxf.Tags method), 140
 - get_flag_state() (DXFObject method), 114
 - get_flag_state() (GraphicEntity method), 64
 - get_geodata() (Modelspace method), 62
 - get_gradient() (Hatch method), 86
 - get_handle() (ezdxf.lldxf.Tags method), 139
 - get_image_def() (Image method), 97
 - get_layout() (GraphicEntity method), 64
 - get_mesh_data() (GeoData method), 117
 - get_mesh_vertex() (Polymesh method), 73
 - get_mesh_vertex_cache() (Polymesh method), 73
 - get_mode() (Polyline method), 71
 - get_name() (DXFGroup method), 112
 - get_ocs() (GraphicEntity method), 64
 - get_paper_limits() (Layout method), 57
 - get_path_entity_transformation_matrix() (ExtrudedSurface method), 101
 - get_path_entity_transformation_matrix() (SweptSurface method), 103
 - get_points() (LWPPolyline method), 79
 - get_pos() (Attdef method), 110
 - get_pos() (Attrib method), 111
 - get_pos() (Text method), 69
 - get_redraw_order() (Layout method), 61
 - get_rotation() (MText method), 82
 - get_row() (ezdxf.algebra.Matrix44 method), 133
 - get_seed_points() (Hatch method), 86
 - get_shx() (StyleTable method), 45

- get_sweep_entity_transformation_matrix() (Extruded-Surface method), 101
 get_sweep_entity_transformation_matrix() (SweptSurface method), 103
 get_text() (MText method), 82
 get_transformation_matrix_extruded_entity() (Extruded-Surface method), 101
 get_transformation_matrix_lofted_entity() (LoftedSurface method), 102
 get_transformation_matrix_path_entity() (SweptSurface method), 103
 get_transformation_matrix_revolved_entity() (RevolvedSurface method), 102
 get_transformation_matrix_sweep_entity() (SweptSurface method), 103
 get_transpose() (ezdxf.algebra.Matrix44 method), 135
 get_underlay_def() (Underlay method), 105
 GradientData (built-in class), 91
 GraphicEntity (built-in class), 63
 grid() (Insert method), 108
 grid_on (VPort.dxf attribute), 53
 grid_spacing (VPort.dxf attribute), 53
 groupby() (EntityQuery method), 120
 groupby() (Layout method), 58
 groups (Drawing attribute), 40
 groups() (DXFGroupTable method), 113
- ## H
- halign (AttDef.dxf attribute), 109
 halign (Attrib.dxf attribute), 111
 halign (Text.dxf attribute), 68
 handle (AppID.dxf attribute), 54
 handle (BlockRecord.dxf attribute), 55
 handle (DimStyle.dxf attribute), 48
 handle (DXFObject.dxf attribute), 114
 handle (GraphicEntity.dxf attribute), 64, 65
 handle (Layer.dxf attribute), 46
 handle (Style.dxf attribute), 47
 handle (UCS.dxf attribute), 55
 handle (View.dxf attribute), 54
 handle (VPort.dxf attribute), 53
 handles() (DXFGroup method), 112
 has_attdef() (BlockLayout method), 62
 has_attr() (Insert method), 108
 has_entry() (Table method), 45
 has_gradient_fill (Hatch attribute), 85
 has_pattern_fill (Hatch attribute), 85
 has_solid_fill (Hatch attribute), 85
 has_tag() (CustomVars method), 44
 has_tag() (ezdxf.lldxf.Tags method), 140
 Hatch (built-in class), 84
 hatch_style (Hatch.dxf attribute), 84
 header (Drawing attribute), 40
 HeaderSection (built-in class), 43
 height (AttDef.dxf attribute), 109
 height (Attrib.dxf attribute), 111
 height (Style.dxf attribute), 47
 height (Text.dxf attribute), 68
 height (View.dxf attribute), 54
 height (VPort.dxf attribute), 53
 history (3DSolid.dxf attribute), 99
 horizontal_unit_scale (GeoData.dxf attribute), 116
 horizontal_units (GeoData.dxf attribute), 116
- ## I
- Image (built-in class), 96
 image_def (Image.dxf attribute), 96
 image_size (Image.dxf attribute), 96
 image_size (ImageDef.dxf attribute), 115
 ImageDef (built-in class), 114
 import_all() (Importer method), 119
 import_blocks() (Importer method), 118
 import_modelspace_entities() (Importer method), 119
 import_table() (Importer method), 118
 import_tables() (Importer method), 118
 Importer (built-in class), 117
 indexed_faces() (Polyface method), 74
 indices (Face attribute), 75
 insert (AttDef.dxf attribute), 109
 insert (Attrib.dxf attribute), 110
 Insert (built-in class), 107
 insert (Image.dxf attribute), 96
 insert (Insert.dxf attribute), 107
 insert (MText.dxf attribute), 81
 insert (Shape.dxf attribute), 69
 insert (Text.dxf attribute), 68
 insert (underlay.dxf attribute), 104
 insert() (LWPolyline method), 80
 insert_vertices() (Polyline method), 72
 inverse() (ezdxf.algebra.Matrix44 method), 135
 invisible (GraphicEntity.dxf attribute), 65
 invisible_edge (3DFace.dxf attribute), 77
 is_2d_polyline (Polyline attribute), 71
 is_3d_polyline (Polyline attribute), 71
 is_binary_data_compressed (Drawing attribute), 41
 is_close() (in module ezdxf.algebra), 125
 is_close_points() (in module ezdxf.algebra), 125
 is_closed (Polyline attribute), 71
 is_closed (PolylinePath attribute), 87
 is_compatible() (Importer method), 119
 is_const (Attdef attribute), 110
 is_const (Attrib attribute), 111
 is_counter_clockwise (ArcEdge attribute), 89
 is_counter_clockwise (EllipseEdge attribute), 90
 is_edge_visible() (Face method), 75
 is_frozen() (Layer method), 46
 is_invisibe (Attrib attribute), 111
 is_invisible (Attdef attribute), 110

is_layout_block (Block attribute), 107
 is_layout_block (BlockLayout attribute), 62
 is_locked() (Layer method), 46
 is_m_closed (Polyline attribute), 71
 is_n_closed (Polyline attribute), 71
 is_null (ezdxf.algebra.Vector attribute), 130
 is_off() (Layer method), 47
 is_on() (Layer method), 47
 is_poly_face_mesh (Polyline attribute), 71
 is_polygon_mesh (Polyline attribute), 71
 is_preset (Attdef attribute), 110
 is_preset (Attrib attribute), 111
 is_verify (Attdef attribute), 110
 is_verify (Attrib attribute), 111
 items (Linetype.dxf attribute), 48

K

knot_tolerance (Spline.dxf attribute), 93
 knot_values (Spline attribute), 94
 knot_values (SplineData attribute), 95
 knot_values (SplineEdge attribute), 90
 knot_values() (ezdxf.algebra.BSpline method), 135

L

last_height (Style.dxf attribute), 47
 Layer (built-in class), 45
 layer (GraphicEntity.dxf attribute), 64, 65
 layers (Drawing attribute), 40
 layout (BlockRecord.dxf attribute), 55
 Layout (built-in class), 57
 layout() (Drawing method), 41
 layout_names() (Drawing method), 41
 Layouts (built-in class), 56
 layouts (Drawing attribute), 40
 layouts_and_blocks() (Drawing method), 41
 length (Linetype.dxf attribute), 48
 lens_length (View.dxf attribute), 54
 lens_length (VPort.dxf attribute), 53
 lerp() (ezdxf.algebra.Vector method), 131
 Line (built-in class), 66
 line_spacing_factor (MText.dxf attribute), 82
 line_spacing_style (MText.dxf attribute), 82
 line_weight (Layer.dxf attribute), 46
 LineEdge (built-in class), 89
 lines (PatternData attribute), 90
 Linetype (built-in class), 48
 linetype (GraphicEntity.dxf attribute), 64, 65
 linetype (Layer.dxf attribute), 46
 linetypes (Drawing attribute), 41
 lineweight (GraphicEntity.dxf attribute), 65
 lineweight (Layer.dxf attribute), 46
 list() (ezdxf.algebra.Vector method), 131
 loaded (ImageDef.dxf attribute), 115
 location (Point.dxf attribute), 66

location (Vertex.dxf attribute), 72
 lock() (Layer method), 46
 LoftedSurface (built-in class), 101
 lower_left (VPort.dxf attribute), 53
 Itscale (GraphicEntity.dxf attribute), 65
 LWPPolyline (built-in class), 78

M

m_close() (Polyline method), 71
 m_count (Polyline.dxf attribute), 70
 m_smooth_density (Polyline.dxf attribute), 70
 magnitude (ezdxf.algebra.Vector attribute), 130
 magnitude_square (ezdxf.algebra.Vector attribute), 130
 major_axis (Ellipse.dxf attribute), 77
 major_axis_vector (EllipseEdge attribute), 90
 material (Layer.dxf attribute), 46
 Matrix44 (class in ezdxf.algebra), 132
 max_t (ezdxf.algebra.BSpline attribute), 135
 MengerSponge (class in ezdxf.addons), 145
 Mesh (built-in class), 91
 mesh() (ezdxf.addons.MengerSponge method), 146
 mesh() (ezdxf.addons.SierpinskyPyramid method), 147
 mesh_faces_count (GeoData.dxf attribute), 117
 MeshBuilder (class in ezdxf.addons), 142
 MeshData (built-in class), 92
 MeshVertexCache (built-in class), 74
 MeshVertexMerger (class in ezdxf.addons), 143
 minor_axis_length (EllipseEdge attribute), 90
 Modelspace (built-in class), 62
 modelspace() (Drawing method), 41
 modelspace() (Layouts method), 56
 monochrome (Underlay attribute), 105
 move_to_layout() (GraphicEntity method), 63
 MText (built-in class), 81
 MText (class in ezdxf.addons), 150
 MTextData (built-in class), 83

N

n_close() (Polyline method), 71
 n_control_points (Spline.dxf attribute), 93
 n_count (Polyline.dxf attribute), 70
 n_fit_points (Spline.dxf attribute), 93
 n_knots (Spline.dxf attribute), 93
 n_seed_points (Hatch.dxf attribute), 84
 n_smooth_density (Polyline.dxf attribute), 71
 name (AppID.dxf attribute), 55
 name (Block.dxf attribute), 106
 name (BlockLayout attribute), 62
 name (BlockRecord.dxf attribute), 55
 name (DimStyle.dxf attribute), 48
 name (Insert.dxf attribute), 107
 name (Layer.dxf attribute), 46
 name (Linetype.dxf attribute), 48
 name (Shape.dxf attribute), 69

name (Style.dxf attribute), 47
 name (UCS.dxf attribute), 55
 name (UnderlayDefinition.dxf attribute), 115
 name (View.dxf attribute), 54
 name (VPort.dxf attribute), 53
 name2 (Block.dxf attribute), 106
 names() (Layouts method), 56
 names_in_taborder() (Layouts method), 56
 new() (BlocksSection method), 105
 new() (DXFGroupTable method), 113
 new() (ezdxf.query method), 121
 new() (Layouts method), 56
 new() (Table method), 44
 new_anonymous_block() (BlocksSection method), 106
 new_geodata() (Modelspace method), 62
 new_layout() (Drawing method), 41
 new_line() (PatternData method), 90
 no_twist (LoftedSurface.dxf attribute), 101
 normalize() (ezdxf.algebra.Vector method), 131
 north_direction (GeoData.dxf attribute), 116

O

oblique (AttDef.dxf attribute), 109
 oblique (Attrib.dxf attribute), 111
 oblique (Shape.dxf attribute), 69
 oblique (Style.dxf attribute), 47
 oblique (Text.dxf attribute), 68
 observation_from_tag (GeoData.dxf attribute), 117
 observation_to_tag (GeoData.dxf attribute), 117
 OCS (class in ezdxf.algebra), 127
 off() (Layer method), 47
 offset (PatternDefinitionLine. attribute), 91
 on (Underlay attribute), 105
 on() (Layer method), 47
 one_color (GradientData attribute), 91
 optimize() (MeshData method), 92
 optimize() (Polyface method), 74
 order (ezdxf.algebra.BSpline attribute), 135
 origin (UCS.dxf attribute), 55
 orthogonal() (ezdxf.algebra.Vector method), 131
 owner (AppID.dxf attribute), 54
 owner (BlockRecord.dxf attribute), 55
 owner (DimStyle.dxf attribute), 48
 owner (DXFObject.dxf attribute), 114
 owner (GraphicEntity.dxf attribute), 65
 owner (Layer.dxf attribute), 46
 owner (Linetype.dxf attribute), 48
 owner (Style.dxf attribute), 47
 owner (UCS.dxf attribute), 55
 owner (View.dxf attribute), 54
 owner (VPort.dxf attribute), 53

P

page_setup() (Layout method), 57

Paperspace (built-in class), 62
 paperspace (GraphicEntity.dxf attribute), 64, 65
 path_entity_id (SweptSurface.dxf attribute), 103
 path_entity_transform_computed (ExtrudedSurface.dxf attribute), 100
 path_entity_transform_computed (SweptSurface.dxf attribute), 103
 path_type_flags (EdgePath attribute), 88
 path_type_flags (PolylinePath attribute), 87
 paths (BoundaryPathData attribute), 87
 pattern_angle (Hatch.dxf attribute), 84
 pattern_double (Hatch.dxf attribute), 84
 pattern_name (Hatch.dxf attribute), 84
 pattern_scale (Hatch.dxf attribute), 84
 pattern_type (Hatch.dxf attribute), 84
 PatternData (built-in class), 90
 PatternDefinitionLine (built-in class), 91
 periodic (SplineEdge attribute), 90
 perspective_projection() (ezdxf.algebra.Matrix44 method), 134
 perspective_projection_fov() (ezdxf.algebra.Matrix44 method), 134
 pixel_size (ImageDef.dxf attribute), 115
 place() (Insert method), 108
 plane_normal_lofting_type (LoftedSurface.dxf attribute), 101
 plot (Layer.dxf attribute), 46
 plot_style_name (Layer.dxf attribute), 46
 Point (built-in class), 66
 point() (ezdxf.algebra.BSpline method), 136
 point() (ezdxf.algebra.DBspline method), 136
 points() (Face method), 75
 points() (LWPolyline method), 80
 points() (Polyline method), 72
 points_from_wcs() (ezdxf.algebra.OCS method), 127
 points_from_wcs() (ezdxf.algebra.UCS method), 128
 points_to_wcs() (ezdxf.algebra.OCS method), 127
 points_to_wcs() (ezdxf.algebra.UCS method), 128
 Polyface (built-in class), 74
 Polyline (built-in class), 70
 PolylinePath (built-in class), 87
 Polymesh (built-in class), 73
 project() (ezdxf.algebra.Vector method), 131
 prompt (AttDef.dxf attribute), 109
 properties (CustomVars attribute), 43
 pyramids() (ezdxf.addons.SierpinskyPyramid method), 147

Q

groupby() (Drawing method), 43
 query() (Drawing method), 42
 query() (EntityQuery method), 120
 query() (Layout method), 58

R

R12FastStreamWriter (built-in class), 123
 r12writer() (built-in function), 123
 radius (Arc.dxf attribute), 67
 radius (ArcEdge attribute), 89
 radius (Circle.dxf attribute), 67
 radius (EllipseEdge attribute), 90
 radius (ezdxf.algebra.Arc attribute), 136
 ratio (Ellipse.dxf attribute), 77
 rational (SplineEdge attribute), 90
 Ray (built-in class), 97
 reference_point (GeoData.dxf attribute), 116
 reference_vector_for_controlling_twist (ExtrudedSurface.dxf attribute), 100
 reference_vector_for_controlling_twist (SweptSurface.dxf attribute), 103
 Region (built-in class), 98
 remove() (CustomVars method), 44
 remove() (EntityQuery method), 120
 remove() (Table method), 44
 remove_invalid_handles() (DXFGroup method), 113
 remove_tags() (ezdxf.lldxf.Tags method), 140
 remove_tags_except() (ezdxf.lldxf.Tags method), 140
 rename() (Layouts method), 56
 rename_block() (BlocksSection method), 106
 render() (ezdxf.addons.Bezier method), 150
 render() (ezdxf.addons.MengerSponge method), 146
 render() (ezdxf.addons.MeshBuilder method), 142
 render() (ezdxf.addons.R12Spline method), 149
 render() (ezdxf.addons.SierpinskyPyramid method), 146
 render_as_fit_points() (ezdxf.addons.Spline method), 147
 render_closed_bspline() (ezdxf.addons.Spline method), 148
 render_closed_rbspline() (ezdxf.addons.Spline method), 148
 render_open_bspline() (ezdxf.addons.Spline method), 147
 render_open_rbspline() (ezdxf.addons.Spline method), 148
 render_polyline() (ezdxf.addons.EulerSpiral method), 150
 render_spline() (ezdxf.addons.EulerSpiral method), 150
 render_uniform_rbspline() (ezdxf.addons.Spline method), 148
 replace() (CustomVars method), 44
 replace() (ezdxf.algebra.Vector method), 131
 replace_handle() (ezdxf.lldxf.Tags method), 139
 reset_boundary() (Image method), 97
 reset_boundary() (Underlay method), 105
 reset_extends() (Layout method), 57
 reset_paper_limits() (Layout method), 57
 reset_viewports() (Layout method), 57
 resolution_units (ImageDef.dxf attribute), 115
 reversed() (ezdxf.algebra.Vector method), 132

revolve_angle (RevolvedSurface.dxf attribute), 102
 RevolvedSurface (built-in class), 102
 rgb (GraphicEntity attribute), 63
 rot_z_deg() (ezdxf.algebra.Vector method), 132
 rot_z_rad() (ezdxf.algebra.Vector method), 132
 rotation (AttDef.dxf attribute), 109
 rotation (Attrib.dxf attribute), 111
 rotation (GradientData attribute), 91
 rotation (Insert.dxf attribute), 107
 rotation (MText.dxf attribute), 82
 rotation (Shape.dxf attribute), 69
 rotation (Text.dxf attribute), 68
 rotation (underlay.dxf attribute), 104
 rotation_form() (in module ezdxf.addons), 145
 row_count (Insert.dxf attribute), 107
 row_spacing (Insert.dxf attribute), 107
 rows() (ezdxf.algebra.Matrix44 method), 135
 ruled_surface (LoftedSurface.dxf attribute), 101

S

save() (Drawing method), 42
 saveas() (Drawing method), 42
 scale (Underlay attribute), 105
 scale() (ezdxf.addons.MeshBuilder method), 142
 scale() (ezdxf.algebra.Matrix44 method), 133
 scale_estimation_method (GeoData.dxf attribute), 116
 scale_factor (ExtrudedSurface.dxf attribute), 100
 scale_factor (SweptSurface.dxf attribute), 103
 scale_x (underlay.dxf attribute), 104
 scale_y (underlay.dxf attribute), 104
 scale_z (underlay.dxf attribute), 104
 sea_level_correction (GeoData.dxf attribute), 116
 sea_level_elevation (GeoData.dxf attribute), 116
 sections (Drawing attribute), 40
 set() (ezdxf.algebra.Matrix44 method), 133
 set_acis_data() (3DSolid method), 99
 set_acis_data() (Body method), 98
 set_acis_data() (Region method), 99
 set_acis_data() (Surface method), 100
 set_align() (Attdef method), 110
 set_align() (Attrib method), 112
 set_align() (Text method), 69
 set_boundary() (Image method), 97
 set_boundary() (Underlay method), 105
 set_col() (ezdxf.algebra.Matrix44 method), 133
 set_color() (Layer method), 47
 set_color() (MTextData method), 83
 set_control_points() (Spline method), 94
 set_coordinate_system_definition() (GeoData method), 117
 set_data() (DXFGroup method), 112
 set_dxf_attrib() (DXFObject method), 114
 set_dxf_attrib() (GraphicEntity method), 64
 set_first() (ezdxf.lldxf.Tags method), 140

- set_fit_points() (Spline method), 94
- set_flag_state() (DXFObject method), 114
- set_flag_state() (GraphicEntity method), 64
- set_font() (MTextData method), 83
- set_gradient() (Hatch method), 85
- set_knot_values() (Spline method), 94
- set_location() (MText method), 82
- set_mesh_data() (GeoData method), 117
- set_mesh_vertex() (Polymesh method), 73
- set_open_rational() (Spline method), 94
- set_open_uniform() (Spline method), 94
- set_path_entity_transformation_matrix() (ExtrudedSurface method), 101
- set_path_entity_transformation_matrix() (SweptSurface method), 103
- set_pattern_definition() (Hatch method), 85
- set_pattern_fill() (Hatch method), 86
- set_periodic() (Spline method), 94
- set_periodic_rational() (Spline method), 95
- set_plot_style() (Layout method), 57
- set_plot_type() (Layout method), 57
- set_plot_window() (Layout method), 57
- set_points() (LWPPolyline method), 79
- set_pos() (Attdef method), 110
- set_pos() (Attrib method), 111
- set_pos() (Text method), 68
- set_redraw_order() (Layout method), 61
- set_rotation() (MText method), 82
- set_row() (ezdxf.algebra.Matrix44 method), 133
- set_seed_points() (Hatch method), 86
- set_solid_fill() (Hatch method), 85
- set_sweep_entity_transformation_matrix() (ExtrudedSurface method), 101
- set_sweep_entity_transformation_matrix() (SweptSurface method), 103
- set_text() (MText method), 82
- set_transformation_matrix_extruded_entity() (ExtrudedSurface method), 101
- set_transformation_matrix_lofted_entity() (LoftedSurface method), 102
- set_transformation_matrix_path_entity() (SweptSurface method), 103
- set_transformation_matrix_revolved_entity() (RevolvedSurface method), 102
- set_transformation_matrix_sweep_entity() (SweptSurface method), 103
- set_uniform() (Spline method), 94
- set_uniform_rational() (Spline method), 95
- set_vertices() (PolylinePath method), 88
- set_weights() (Spline method), 94
- shadow_mode (GraphicEntity.dxf attribute), 65
- Shape (built-in class), 69
- SierpinskyPyramid (class in ezdxf.addons), 146
- simple_surfaces (LoftedSurface.dxf attribute), 101
- size (Shape.dxf attribute), 69
- smooth_type (Polyline.dxf attribute), 71
- snap_base (VPort.dxf attribute), 53
- snap_isopair (VPort.dxf attribute), 53
- snap_on (VPort.dxf attribute), 53
- snap_rotation (VPort.dxf attribute), 53
- snap_spacing (VPort.dxf attribute), 53
- snap_style (VPort.dxf attribute), 53
- Solid (built-in class), 75
- solid (ExtrudedSurface.dxf attribute), 100
- solid (LoftedSurface.dxf attribute), 101
- solid (RevolvedSurface.dxf attribute), 102
- solid (SweptSurface.dxf attribute), 103
- solid_fill (Hatch.dxf attribute), 84
- source_boundary_objects (EdgePath attribute), 88
- source_boundary_objects (PolylinePath attribute), 87
- spatial_angle_deg (ezdxf.algebra.Vector attribute), 130
- spatial_angle_rad (ezdxf.algebra.Vector attribute), 130
- Spline (built-in class), 93
- Spline (class in ezdxf.addons), 147
- SplineData (built-in class), 95
- SplineEdge (built-in class), 90
- start (Line.dxf attribute), 66
- start (LineEdge attribute), 89
- start (Ray.dxf attribute), 97
- start (XLine.dxf attribute), 98
- start() (ezdxf.addons.Bezier method), 149
- start_angle (Arc.dxf attribute), 67
- start_angle (ArcEdge attribute), 89
- start_angle (EllipseEdge attribute), 90
- start_angle (ezdxf.algebra.Arc attribute), 136
- start_angle (RevolvedSurface.dxf attribute), 102
- start_angle_rad (ezdxf.algebra.Arc attribute), 137
- start_draft_angle (LoftedSurface.dxf attribute), 101
- start_draft_distance (RevolvedSurface.dxf attribute), 102
- start_draft_magnitude (LoftedSurface.dxf attribute), 101
- start_param (Ellipse.dxf attribute), 78
- start_tangent (Spline.dxf attribute), 94
- start_tangent (SplineEdge attribute), 90
- start_width (Vertex.dxf attribute), 72
- status (VPort.dxf attribute), 53
- strip() (ezdxf.lldxf.Tags method), 139
- style (AttDef.dxf attribute), 109
- style (Attrib.dxf attribute), 111
- Style (built-in class), 47
- style (MText.dxf attribute), 82
- style (Text.dxf attribute), 68
- styles (Drawing attribute), 40
- StyleTable (built-in class), 45
- subdivision_levels (Mesh.dxf attribute), 92
- supported_dxf_attrib() (DXFObject method), 114
- supported_dxf_attrib() (GraphicEntity method), 64
- Surface (built-in class), 99
- sweep_alignment (SweptSurface.dxf attribute), 103

sweep_alignment_flags (ExtrudedSurface.dxf attribute), 100
 sweep_entity_transform_computed (ExtrudedSurface.dxf attribute), 100
 sweep_entity_transform_computed (SweptSurface.dxf attribute), 103
 sweep_vector (ExtrudedSurface.dxf attribute), 100
 swept_entity_id (SweptSurface.dxf attribute), 103
 SweptSurface (built-in class), 102

T

Table (built-in class), 44
 Table (class in ezdxf.addons), 151
 tag (AttDef.dxf attribute), 109
 tag (Attrib.dxf attribute), 110
 tag_index() (ezdxf.lldxf.Tags method), 140
 TagArray (class in ezdxf.lldxf), 141
 TagDict (class in ezdxf.lldxf), 141
 TagList (class in ezdxf.lldxf), 141
 Tags (class in ezdxf.lldxf), 139
 tangent (Vertex.dxf attribute), 73
 target_point (View.dxf attribute), 54
 target_point (VPort.dxf attribute), 53
 templatedir (ezdxf.options attribute), 39
 text (AttDef.dxf attribute), 109
 text (Attrib.dxf attribute), 110
 Text (built-in class), 67
 text (MTextData attribute), 83
 text (Text.dxf attribute), 67
 text_direction (MText.dxf attribute), 82
 text_generation_flag (AttDef.dxf attribute), 109
 text_generation_flag (Attrib.dxf attribute), 111
 text_generation_flag (Text.dxf attribute), 68
 text_generation_flags (Style.dxf attribute), 47
 text_lines (ModelerGeometryData attribute), 98
 thaw() (Layer method), 46
 thickness (GraphicEntity.dxf attribute), 65
 tint (GradientData attribute), 91
 to_ocs() (ezdxf.algebra.UCS method), 128
 to_wcs() (ezdxf.algebra.OCS method), 127
 to_wcs() (ezdxf.algebra.UCS method), 128
 Trace (built-in class), 76
 transform() (ezdxf.addons.MeshBuilder method), 142
 transform() (ezdxf.algebra.Matrix44 method), 135
 transform_vectors() (ezdxf.algebra.Matrix44 method), 135
 translate() (ezdxf.addons.MeshBuilder method), 142
 translate() (ezdxf.algebra.Matrix44 method), 133
 transparency (GraphicEntity attribute), 63
 transparency (GraphicEntity.dxf attribute), 65
 transpose() (ezdxf.algebra.Matrix44 method), 135
 true_color (GraphicEntity.dxf attribute), 65
 twist_angle (ExtrudedSurface.dxf attribute), 100
 twist_angle (RevolvedSurface.dxf attribute), 102

twist_angle (SweptSurface.dxf attribute), 103

U

u_count (Surface.dxf attribute), 99
 u_pixel (Image.dxf attribute), 96
 UCS (built-in class), 55
 UCS (class in ezdxf.algebra), 127
 ucs (Drawing attribute), 41
 ucs() (UCS method), 55
 ucs_icon (VPort.dxf attribute), 53
 Underlay (built-in class), 104
 underlay_def (underlay.dxf attribute), 104
 UnderlayDefinition (built-in class), 115
 unit_vector (Ray.dxf attribute), 97
 unit_vector (XLine.dxf attribute), 98
 unlink_entity() (Layout method), 62
 unlock() (Layer method), 46
 up_direction (GeoData.dxf attribute), 116
 update() (ezdxf.lldxf.Tags method), 140
 update_attribs() (DXFObject method), 114
 update_attribs() (GraphicEntity method), 64
 upper_right (VPort.dxf attribute), 53
 user_scale_factor (GeoData.dxf attribute), 116

V

v_count (Surface.dxf attribute), 99
 v_pixel (Image.dxf attribute), 96
 valid_dxf_attrib_names() (DXFObject method), 114
 valid_dxf_attrib_names() (GraphicEntity method), 64
 valign (AttDef.dxf attribute), 109
 valign (Attrib.dxf attribute), 111
 valign (Text.dxf attribute), 68
 value (ezdxf.lldxf.DXFTag attribute), 138
 value (ezdxf.lldxf.DXFVertex attribute), 139
 Vector (class in ezdxf.algebra), 129
 version (GeoData.dxf attribute), 116
 version (Mesh.dxf attribute), 92
 Vertex (built-in class), 72
 VertexArray (class in ezdxf.lldxf), 141
 vertical_unit_scale (GeoData.dxf attribute), 116
 vertical_units (GeoData.dxf attribute), 116
 vertices (Face attribute), 75
 vertices (MeshData attribute), 92
 vertices (MeshVertexCache attribute), 74
 vertices (PolylinePath attribute), 87
 vertices() (LWPolyline method), 80
 vertices() (Polyline method), 72
 vertices_in_wcs() (LWPolyline method), 80
 View (built-in class), 54
 view_mode (View.dxf attribute), 54
 view_mode (VPort.dxf attribute), 53
 view_twist (View.dxf attribute), 54
 view_twist (VPort.dxf attribute), 53
 viewports (Drawing attribute), 41

ViewportTable (built-in class), 45
 views (Drawing attribute), 41
 virtual_guide (LoftedSurface.dxf attribute), 101
 VPort (built-in class), 53
 vtx0 (3DFace.dxf attribute), 77
 vtx0 (Solid.dxf attribute), 75
 vtx0 (Trace.dxf attribute), 76
 vtx1 (3DFace.dxf attribute), 77
 vtx1 (Solid.dxf attribute), 76
 vtx1 (Trace.dxf attribute), 76
 vtx1 (Vertex.dxf attribute), 73
 vtx2 (3DFace.dxf attribute), 77
 vtx2 (Solid.dxf attribute), 76
 vtx2 (Trace.dxf attribute), 76
 vtx2 (Vertex.dxf attribute), 73
 vtx3 (3DFace.dxf attribute), 77
 vtx3 (Solid.dxf attribute), 76
 vtx3 (Trace.dxf attribute), 76
 vtx3 (Vertex.dxf attribute), 73
 vtx4 (Vertex.dxf attribute), 73

z_rotate() (ezdxf.algebra.Matrix44 method), 133
 zscale (Insert.dxf attribute), 107

W

weights (Spline attribute), 94
 weights (SplineData attribute), 95
 weights (SplineEdge attribute), 90
 width (AttDef.dxf attribute), 109
 width (Attrib.dxf attribute), 111
 width (MText.dxf attribute), 81
 width (Style.dxf attribute), 47
 width (Text.dxf attribute), 68
 width (View.dxf attribute), 54
 write() (Drawing method), 42

X

x (ezdxf.algebra.Vector attribute), 130
 x_rotate() (ezdxf.algebra.Matrix44 method), 133
 xaxis (UCS.dxf attribute), 55
 XLine (built-in class), 97
 xref_path (Block.dxf attribute), 107
 xscale (Insert.dxf attribute), 107
 xscale (Shape.dxf attribute), 69
 xy (ezdxf.algebra.Vector attribute), 130
 xyz (ezdxf.algebra.Vector attribute), 130
 xyz_rotate() (ezdxf.algebra.Matrix44 method), 134

Y

y (ezdxf.algebra.Vector attribute), 130
 y_rotate() (ezdxf.algebra.Matrix44 method), 133
 yaxis (UCS.dxf attribute), 55
 yscale (Insert.dxf attribute), 107

Z

z (ezdxf.algebra.Vector attribute), 130