ezdxf Documentation

Release 0.8.1

Manfred Moitzi

Apr 06, 2017

Contents

1 Quick-Info **3**

2 Contents **5**

2.1 Introduction 5
2.2 Tutorials 6
2.3 Reference 23
2.4 Howto 79
2.5 DXF Internals 81
2.6 News 83

3 Indices and tables **87**

Welcome! This is the documentation for ezdxf 0.8.1, last updated Apr 06, 2017.

CHAPTER 1

Quick-Info

- *ezdxf* is a Python package to read and write DXF drawings (interface to the DXF file format)
- intended audience: Developer
- requires Python 2.7 or later, runs on CPython and pypy, maybe on IronPython and Jython
- OS independent
- additional packages required: pyparsing
- MIT-License
- supported DXF versions read/new: R12, R2000, R2004, R2007, R2010 and R2013
- support for DXF versions R13/R14 (AC1012/AC1014), will be upgraded to R2000 (AC1015)
- support for older versions than R12, will be upgraded to R12 (AC1009)
- preserves third-party DXF content
- additional *Fast DXF R12 File/Stream Writer*, just the ENTITIES section with support for LINE, CIRCLE, ARC, TEXT, POINT, SOLID, 3DFACE and POLYLINE.

CHAPTER 2

Contents

Introduction

What is ezdxf

ezdxf is a Python package which allows developers to read 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.

ezdxf is still in its infancy, therefore not all DXF features supported yet, but additional features will be added in the future gradually.

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 (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.

Supported Python Versions

ezdxf requires at least Python 2.7 and it's Python 3 compatible. I run unit tests with CPython 2.7, the latest stable CPython 3 version and the latest stable release of pypy during development. *ezdxf* is written in pure Python and

requires only *pyparser* as additional library beside the Python Standard Library, hence it should run with IronPython and Jython also.

Supported Operating Systems

ezdxf is OS independent and runs on all platforms which provide an appropriate Python interpreter (>=2.7).

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

ezdxf reads also older versions but saves it as R12.

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.

License

ezdxf is licensed under the very liberal MIT-License.

Tutorials

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 construction element like LINE, CIRCLE, TEXT and so on. Every construction element has to reside in exact one layout.

There are three different layout types:

- model space: this is the common construction place
- paper space: used to to create printable drawings
- 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 standards can have more than one paper space and each paper space has a name.

Iterate over DXF Entities of a Layout

Iterate over all construction elements in the model space:

```
modelspace = dwg.modelspace()
for e in modelspace:
    if e.dxftype() == 'LINE':
        print("LINE on layer: %s\n" % e.dxf.layer)
        print("start point: %s\n" % e.dxf.start)
        print("end point: %s\n" % e.dxf.end)
```

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

Access DXF Attributes of an Entity

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

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()*.

Iterate Over All DXF Entities at Once

Because the DXF entities of the model space and the entities of all paper spaces are stored in the ENTITIES section of the DXF drawing, you can also iterate over all drawing elements at once, except the entities placed in the block layouts:

```
for e in dwg.entities:
    print("DXF Entity: %s\n" % e.dxftype())
```

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
↪"]')
```

Default Layer Settings

See also:

Tutorial for Layers and class `Layer`

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.

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 ans *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*.

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
```

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})

# 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)
```

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, ent_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)]
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.

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.
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 *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.

more is coming soon ...

Tutorial for MText

coming soon ...

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 manger
    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())
```

Tutorial for Polyface

coming soon ...

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")
```

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

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

Tutorial for Hatch Pattern Definition

TODO

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")
```

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,
# 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")
```

Tutorial for Linetypes

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

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

total_pattern_length Sum of all linetype elements (absolute vaues)

elem if eleme > 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

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 still used linetypes leads to an 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.

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.

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`**(*dxfversion='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`. *dxfversion* 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

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'*)

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).

ezdxf.**read**(*stream*)
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'.

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 binray data for every drawing you open, but data compression cost 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.**compress_default_chunks**
There are at least two sections in DXF drawings which are very useless: *THUMBNAILIMAGE* and since AutoCAD 2013 (AC1027) *ACDSDATA*. They were managed by the simple `DefaultChunk` class, which is just a bunch of dumb tags, to save some memory you can compress these default chunks by setting the option `ezdxf.options.compress_default_chunks = True`.

ezdxf.options.**templatedir**
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.**store_comments**

- preserves the existing comments at the top of the file
- adds a comment when upgrading the DXF version
- adds a *'last saved by ezdxf ...'* comment

Default setting 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.`**`dxffactory`**
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.`**`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.**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.

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 *KeyError* if *handle* doesn't exist. Returns `DXFEntity` or inherited.

If you just need the raw DXF tags use:

```
tags = Drawing.entitydb[handle]  # raises KeyError, 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: `ClassifiedTags`

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. 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 `ValueError` 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 `ValueError` if *tag* does not exist.

Tables

Table Class

class Table

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

Parameters

- **name** (*str*) – name of the new table-entry
- **dxfattribs** (*dict*) – 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 *ValueError* if table-entry is not present.

Table.**remove**(*name*)
Removes table-entry *name*. Raises *ValueError* if table-entry is not present.

Table.**__len__**()
Get count of table-entries.

Table.**__contains__**(*name*)
True if table contains a table-entry *name*.

Table.**__iter__**()
Iterate over all table.entries, yields table-entry-objects.

Table Entry Classes

Layer

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

`Layer.`**dxf**
The DXF attributes namespace, access DXF attributes by this attribute, like `object.dxf.linetype = 'DASHED'`. Just the *dxf* attribute is read only, the DXF attributes are read- and writeable. (read only)

DXFAttr	Version	Description
handle	R12	DXF handle (feature for experts)
name	R12	layer name (str)
flags	R12	layer flags (feature for experts)
color	R12	layer color, but use *Layer.get_color()*, because color is negative for layer status *off* (int)
linetype	R12	name of line type (str)
plot	R13	plot flag (int), `1` for plot layer (default value), `0` for don't plot layer
line_weight	R13	line weight enum value (int)
plot_style_name	R13	handle to PlotStyleName (feature for experts)

`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*

`Style.`**`dxf`**
The DXF attributes namespace.

DXFAttr	Description
handle	DXF handle (feature for experts)
name	style name (str)
flags	layer flags (feature for experts)
height	fixed height in drawing units, `0` for not fixed (float)
width	width factor (float), default is `1`
oblique	oblique angle in degrees, `0` is vertical (float)
text_generation_flags	**text generations flags (int)** • 2 = text is backward (mirrored in X) • 4 = text is upside down (mirrored in Y)
last_height	last height used in drawing units (float)
font	primary font file name (str)
bigfont	big font name, blank if none (str)

Linetype

class `Linetype`
Defines a linetype.

`Linetype.`**`dxf`**
The DXF attributes namespace.

DXFAttr	Description
name	linetype name (str)
description	linetype description (str)
length	total pattern length in drawing units (float)
items	number of linetype elements (int)

DimStyle

class `DimStyle`
Defines a dimension style.

`DimStyle.`**`dxf`**
The DXF attributes namespace.

TODO DXFAttr for DimStyle class

Viewport

class `Viewport`
Defines a viewport to the model space.

`Viewport.`**`dxf`**
The DXF attributes namespace.

TODO DXFAttr for the Viewport class

View

class `View`
Defines a view.

`View`.**dxf**
The DXF attributes namespace.

TODO DXFAttr for the View class

AppID

class `AppID`
Defines an AppID.

`AppID`.**dxf**
The DXF attributes namespace.

TODO DXFAttr for the AppID class

UCS

class `UCS`
Defines an user coordinate system (UCS).

`UCS`.**dxf**
The DXF attributes namespace.

TODO DXFAttr for the UCS class

BlockRecord

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

`BlockRecord`.**dxf**
The DXF attributes namespace.

TODO DXFAttr for the BlockRecord class

Layouts

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 layouts 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`

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*, *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.

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)

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.

Warning: Moving DXF entities between layouts is not well tested and may break the DXF structure!

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

Paper Space

class Paperspace
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.**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.

Entities

Common 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.**dxftype**
(read only) Get the DXF type string, like `LINE` for the line entity.

GraphicEntity.**handle**
(read only) Get the entity handle. (feature for experts)

GraphicEntity.**drawing**
(read only) Get the associated drawing.

GraphicEntity.**dxffactory**
(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 `ValueError`), 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 `ValueError`), 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.**get_dxf_attrib**(*key*, *default=ValueError*)
Get DXF attribute *key*, returns *default* if key doesn't exist, or raise `ValueError` if *default* is `ValueError` 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.

Common DXF attributes for DXF R12

Access DXF attributes by the *dxf* attribute of an entity, like `object.dxf.layer = 'MyLayer'`.

DXFAttr	Description
handle	DXF handle (feature for experts)
layer	layer name as string; default=0
linetype	linetype as string, special names BYLAYER, BYBLOCK; default=BYLAYER
color	dxf color index, 0 ... BYBLOCK, 256 ... BYLAYER; default=256
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
extrusion	extrusion direction as 3D point; default=(0, 0, 1)

Common DXF attributes for DXF R13 or later

Access DXF attributes by the *dxf* attribute of an entity, like `object.dxf.layer = 'MyLayer'`.

DXFAttr	Description
handle	DXF handle (feature for experts)
owner	handle to owner, it's a BLOCK_RECORD entry (feature for experts)
layer	layer name as string; default = 0
linetype	linetype as string, special names BYLAYER, BYBLOCK; default=BYLAYER
color	dxf color index, 0 ... BYBLOCK, 256 ... BYLAYER; default= 256
lineweight	lineweight enum value. Stored and moved around as a 16-bit integer.
ltscale	line type scale as float; default=1.0
invisible	1 for invisible, 0 for visible; default=0
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
extrusion	extrusion direction as 3D point; default=(0, 0, 1)
thickness	entity thickness as float; default=0
true_color	true color value as int 0x00RRGGBB, requires DXF Version AC1018 (AutoCAD R2004)
color_name	color name as string, requires DXF Version AC1018 (AutoCAD R2004)
transparency	transparency value as int, 0x020000TT 0x00 = 100% transparent / 0xFF = opaque, requires DXF Version AC1018 (AutoCAD R2004)
shadow_mode	as int; 0 = Casts and receives shadows, 1 = Casts shadows, 2 = Receives shadows, 3 = Ignores shadows; requires DXF Version AC1021 (AutoCAD R2007)

Line

class Line(*GraphicEntity*)

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

DXFAttr	Version	Description
start	R12	start point of line (2D/3D Point)
end	R12	end point of line (2D/3D Point)

Point

class Point(*GraphicEntity*)

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

DXFAttr	Version	Description
location	R12	location of the point (2D/3D Point)

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()`.

DXFAttr	Version	Description
center	R12	center point of circle (2D/3D Point)
radius	R12	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()`.

DXFAttr	Version	Description
center	R12	center point of arc (2D/3D Point)
radius	R12	radius of arc (float)
start_angle	R12	start angle in degrees (float)
end_angle	R12	end angle in degrees (float)

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 2*pi. The ellipse goes from start to end param in *counter clockwise* direction. Create ellipses in layouts and blocks by factory function `add_ellipse()`.

DXFAttr	Version	Description
center	R13	center point of circle (2D/3D Point)
major_axis	R13	Endpoint of major axis, relative to the center (tuple of float)
ratio	R13	Ratio of minor axis to major axis (float)
start_param	R13	Start parameter (this value is 0.0 for a full ellipse) (float)
end_param	R13	End parameter (this value is 2*pi for a full ellipse) (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()`.

DXFAttr	Version	Description
text	R12	the content text itself (str)
insert	R12	first alignment point of text (2D/3D Point), relevant for the adjustments `LEFT`, `ALIGN` and `FIT`.
align_point	R12	second alignment point of text (2D/3D Point), 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`)
height	R12	text height in drawing units (float); default=1
rotation	R12	text rotation in degrees (float); default=0
oblique	R12	text oblique angle (float); default=0
style	R12	text style name (str); default=''STANDARD''
width	R12	width scale factor (float); default=1
halign	R12	horizontal alignment flag (int), use *`Text.set_pos()`* and *`Text.get_align()`*; default=0
valign	R12	vertical alignment flag (int), use *`Text.set_pos()`* and *`Text.get_align()`*; default=0
text_generation_flag	R12	**text generation flags (int)** • 2 = text is backward (mirrored in X) • 4 = text is upside down (mirrored in Y)

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

Parameters

- **`p1`** – first alignment point as (x, y[, z])-tuple
- **`p2`** – second alignment point as (x, y[, z])-tuple, required for `ALIGNED` and `FIT` else ignored
- **`align`** (*`str`*) – 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.

Align-ment	Description
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 *alignpoint* manually.

Polyline

class Polyline(*GraphicEntity*)
The *POLYLINE* entity is very complex, it's use 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().

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

DXFAttr	Ver-sion	Description
elevation	R12	elevation point, the X and Y values are always 0, and the Z value is the polyline's elevation (3D Point)
flags	R12	polyline flags (int), see table below
de-fault_start_width	R12	default line start width (float); default=0
de-fault_end_width	R12	default line end width (float); default=0
m_count	R12	polymesh M vertex count (int); default=1
n_count	R12	polymesh N vertex count (int); default=1
m_smooth_density	R12	smooth surface M density (int); default=0
n_smooth_density	R12	smooth surface N density (int); default=0
smooth_type	R12	Curves and smooth surface type (int); default=0, see table below

Polyline constants for *flags* defined in ezdxf.const:

Polyline.dxf.flags	Value	Description
POLYLINE_CLOSED	1	This is a closed Polyline (or a polygon mesh closed in the M direction)
POLYLINE_MESH_CLOSED_M_DIRECTION	1	equals POLYLINE_CLOSED
POLYLINE_CURVE_FIT_VERTICES_ADDED	2	Curve-fit vertices have been added
POLYLINE_SPLINE_FIT_VERTICES_ADDED	4	Spline-fit vertices have been added
POLYLINE_3D_POLYLINE	8	This is a 3D Polyline
POLYLINE_3D_POLYMESH	16	This is a 3D polygon mesh
POLYLINE_MESH_CLOSED_N_DIRECTION	32	The polygon mesh is closed in the N direction
POLYLINE_POLYFACE_MESH	64	This Polyline is a polyface mesh
POLYLINE_GENERATE_LINETYPE_PATTERN	128	The linetype pattern is generated continuously around the vertices of this Polyline

Polymesh constants for *smooth_type* defined in `ezdxf.const`:

Polyline.dxf.smooth_type	Value	Description
POLYMESH_NO_SMOOTH	0	no smooth surface fitted
POLYMESH_QUADRIC_BSPLINE	5	quadratic B-spline surface
POLYMESH_CUBIC_BSPLINE	6	cubic B-spline surface
POLYMESH_BEZIER_SURFACE	8	Bezier surface

`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.`**`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.

Parameters

- **points** – iterable polyline points, every point is a (x, y[, z])-tuple.
- **dxfattribs** – dict of DXF attributes for the *Vertex*

Polyline.**insert_vertices**(*pos*, *points*, *dxfattribs=None*)
Insert points as *Vertex* objects at position *pos*.

Parameters

- **pos** (*int*) – 0-baesd insert position
- **points** (*iterable*) – iterable polyline points, every point is a tuple.
- **dxfattribs** – dict of DXF attributes for the *Vertex*

Polyline.**delete_vertices**(*pos*, *count=1*)
Delete *count* vertices at position *pos*.

Parameters

- **pos** (*int*) – 0-based insert position
- **count** (*int*) – 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.

DXFAttr	Version	Description
location	R12	vertex location (2D/3D Point)
start_width	R12	line segment start width (float); default=0
end_width	R12	line segment end width (float); default=0
bulge	R12	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.
flags	R12	vertex flags (int), see table below.
tangent	R12	curve fit tangent direction (float)
vtx1	R12	index of 1st vertex, if used as face (feature for experts)
vtx2	R12	index of 2nd vertex, if used as face (feature for experts)
vtx3	R12	index of 3rd vertex, if used as face (feature for experts)
vtx4	R12	index of 4th vertex, if used as face (feature for experts)

Vertex constants for *flags* defined in `ezdxf.const`:

Vertex.dxf.flags	Value	Description
VTX_EXTRA_VERTEX_CREATED	1	Extra vertex created by curve-fitting
VTX_CURVE_FIT_TANGENT	2	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.
VTX_SPLINE_VERTEX_CREATED	8	Spline vertex created by spline-fitting
VTX_SPLINE_FRAME_CONTROL_POINT	16	Spline frame control point
VTX_3D_POLYLINE_VERTEX	32	3D polyline vertex
VTX_3D_POLYGON_MESH_VERTEX	64	3D polygon mesh
VTX_3D_POLYFACE_MESH_VERTEX	128	Polyface mesh vertex

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*.

Parameters **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*.

Parameters

- **pos** – 0-based (row, col)-tuple
- **point** – vertex coordinates as (x, y, z)-tuple
- **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

Parameters **pos** (*tuple*) – 0-based (row, col)-tuple

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

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

Parameters

- **pos** – 0-based (row, col)-tuple
- **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.

Parameters

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

Parameters

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

Parameters **precision** (*int*) – 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*.

Parameters **pos** (*int*) – 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*.

Parameters **pos** (*int*) – 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()*.

DXFAttr	Version	Description
vtx0	R12	location of the 1. point (2D/3D Point)
vtx1	R12	location of the 2. point (2D/3D Point)
vtx2	R12	location of the 3. point (2D/3D Point)
vtx3	R12	location of the 4. point (2D/3D Point)

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()*.

DXFAttr	Version	Description
vtx0	R12	location of the 1. point (2D/3D Point)
vtx1	R12	location of the 2. point (2D/3D Point)
vtx2	R12	location of the 3. point (2D/3D Point)
vtx3	R12	location of the 4. point (2D/3D Point)

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()*.

DXFAttr	Version	Description
vtx0	R12	location of the 1. point (3D Point)
vtx1	R12	location of the 2. point (3D Point)
vtx2	R12	location of the 3. point (3D Point)
vtx3	R12	location of the 4. point (3D Point)
invisible_edge	R12	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.

LWPolyline

class `LWPolyline` (*GraphicEntity*)

Introduced in AutoCAD R13 (DXF version 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()*. LWPolylines are planar elements, therefore all coordinates have no value for the z axis.

See also:

Tutorial for LWPolyline

DXFAttr	Version	Description
elevation	R13	z-axis value in WCS is the polyline elevation (float), default=0
flags	R13	polyline flags (int), see table below
const_width	R13	constant line width (float), default=0
count	R13	number of vertices

LWPolyline constants for *flags* defined in `ezdxf.const`:

LWPolyline.dxf.flags	Value	Description
LWPOLYLINE_CLOSED	1	polyline is closed
LWPOLYLINE_PLINEGEN	128	???

`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.`**`get_points`**`()`

Returns all polyline points as list of tuples (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).

`LWPolyline.`**`get_rstrip_points`**`()`

Generates points without appending zeros: yields (x1, y1), (x2, y2) instead of (x1, y1, 0, 0, 0), (x2, y2, 0, 0, 0).

`LWPolyline.`**`set_points`**`(`*points*`)`

Remove all points and 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).

`LWPolyline.`**`points`**`()`

Context manager for polyline points. Returns a list of tuples (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). Setting/Appending points accepts (x, y, [start_width, [end_width, [bulge]]]) tuples. Set start_width, end_width to 0 to be ignored (x, y, 0, 0, bulge).

`LWPolyline.`**`rstrip_points`**`()`
Context manager for polyline points without appending zeros.

`LWPolyline.`**`append_points`**`(`*points*`)`
Append additional *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).

`LWPolyline.`**`discard_points`**`()`
Remove all points.

`LWPolyline.`**`__len__`**`()`
Number of polyline vertices.

`LWPolyline.`**`__getitem__`**`(`*index*`)`
Get point at position *index* as (x, y, start_width, end_width, bulge) tuple. Actual implementation is very slow! start_width, end_width and bulge is 0 if not present (0 is the DXF default value if not present).

MText

class `MText` (*GraphicEntity*)
Introduced in AutoCAD R13 (DXF version AC1012), extended in AutoCAD 2007 (DXF version 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

DXFAttr	Version	Description
insert	R13	Insertion point (3D Point)
char_height	R13	initial text height (float); default=1.0
width	R13	reference rectangle width (float)
attachment_point	R13	attachment point (int), see table below
flow_direction	R13	text flow direction (int), see table below
style	R13	text style (string); default='STANDARD'
text_direction	R13	x-axis direction vector in WCS (3D Point); default=(1, 0, 0); if *rotation* and *text_direction* are present, *text_direction* wins
rotation	R13	text rotation in degrees (float); default=0
line_spacing_style	R13	line spacing style (int), see table below
line_spacing_factor	R13	percentage of default (3-on-5) line spacing to be applied. Valid values range from 0.25 to 4.00 (float)

MText constants for *attachment_point* 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 constants for *flow_direction* 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 constants for *line_spacing_style* 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.`**`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."
```

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"`)

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()*.

DXFAttr	Version	Description
insert	R12	insertion point as (2D/3D Point)
name	R12	shape name
size	R12	shape size
rotation	R12	rotation angle in degrees; default=0
xscale	R12	relative X scale factor; default=1
oblique	R12	oblique angle; default=0

Ray

class Ray(*GraphicEntity*)
Introduced in AutoCAD R13 (DXF version AC1012), *dxfversion* is `RAY`.

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

DXFAttr	Version	Description
start	R13	start point as (3D Point)
unit_vector	R13	unit direction vector as (3D Point)

XLine

class **XLine** (*GraphicEntity*)

Introduced in AutoCAD R13 (DXF version 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()*.

DXFAttr	Version	Description
start	R13	location point of line as (3D Point)
unit_vector	R13	unit direction vector as (3D Point)

Spline

class **Spline** (*GraphicEntity*)

Introduced in AutoCAD R13 (DXF version 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.

DXFAttr	Version	Description
degree	R13	degree of the spline curve (int)
flags	R13	bit coded option flags (see table below)
n_knots	R13	count of knot values (int), automatically set by *ezdxf*, treat it as read only
n_fit_points	R13	count of fit points (int), automatically set by *ezdxf*, treat it as read only
n_control_points	R13	count of control points (int), automatically set by *ezdxf*, treat it as read only
knot_tolerance	R13	knot tolerance (float); default=1e-10
fit_tolerance	R13	fit tolerance (float); default=1e-10
control_point_tolerance	R13	control point tolerance (float); default=1e-10
start_tangent	R13	start tangent vector as (3D Point)
end_tangent	R13	ene tangent vector as (3D Point)

Spline constants for *flags* 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

See also:

Tutorial for Spline

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.**get_control_points**()
Returns the control points as list of (x, y, z) tuples.

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

Spline.**get_fit_points**()
Returns the fit points as list of (x, y, z) tuples.

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

Spline.**get_knot_values**()
Returns the knot values as list of *floats*.

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

Spline.**get_weights**()
Returns the weight values as list of *floats*.

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

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 standard python lists: 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 calls all spline set methods automatically
```

class SplineData

SplineData.**fit_points**
Standard Python list of *Spline* fit points as (x, y, z)-tuples. (read/write)

SplineData.**control_points**
Standard Python list of *Spline* control points as (x, y, z)-tuples. (read/write)

SplineData.**knot_values**
Standard Python list of *Spline* knot values as floats. (read/write)

SplineData.**weights**
Standard Python list of *Spline* weights as floats. (read/write)

Body

class Body(*GraphicEntity*)
Introduced in AutoCAD R13 (DXF version 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.**text_lines**
ACIS date 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 AutoCAD R13 (DXF version 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 AutoCAD R13 (DXF version 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()*.

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.

DXFAttr	Version	Description
history	R13	handle to history object, see: *Low Level Access to DXF entities*

Image

class `Image`(*GraphicEntity*)

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

DXFAttr	Version	Description
insert	R13	Insertion point, lower left corner of the image
u_pixel	R13	U-vector of a single pixel (points along the visual bottom of the image, starting at the insertion point) (x, y, z) tuple
v_pixel	R13	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_size	R13	Image size in pixels
image_def	R13	Handle to the image definition entity, see `ImageDef`
flags	R13	see table below
clipping	R13	Clipping state: 0 = Off; 1 = On
brightness	R13	Brightness value (0-100; default = 50)
contrast	R13	Contrast value (0-100; default = 50)
fade	R13	Fade value (0-100; default = 0)
clipping_boundary_type	R13	Clipping boundary type. 1 = Rectangular; 2 = Polygonal
count_boundary_points	R13	Number of clip boundary vertices

Image.dxf.flags	Value	Description
IMAGE_SHOW	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_TRANSPARENCY_IS_ON	8	Transparency is on

`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`.

ImageDef

class `ImageDef` (*GraphicEntity*)
Introduced in AutoCAD R13 (DXF version AC1012), *dxftype* is `IMAGEDEF`.

ImageDef defines an image, which can be placed by the *Image* entity. Create *ImageDef* by the *Drawing* factory function *add_image_def()*.

DXFAttr	Version	Description
filename	R13	Relative (to the DXF file) or absolute path to the image file as string
image_size	R13	Image size in pixel as (x, y) tuple
pixel_size	R13	Default size of one pixel in AutoCAD units (x, y) tuple
loaded	R13	Default = 1
resolution_units	R13	Resolution units. 0 = No units; 2 = Centimeters; 5 = Inch, default is 0

Underlay

class `Underlay` (*GraphicEntity*)
Introduced in AutoCAD R13 (DXF version AC1012), *dxftype* 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.

DXFAttr	Version	Description
insert	R13	Insertion point, lower left corner of the image
scale_x	R13	scaling factor in x dircetion (float)
scale_y	R13	scaling factor in y dircetion (float)
scale_z	R13	scaling factor in z dircetion (float)
rotation	R13	ccw rotation in degrees around the extrusion vector (float)
extrusion	R13	extrusion vector (default=0, 0, 1)
underlay_def	R13	Handle to the underlay definition entity, see *UnderlayDefinition*
flags	R13	see table below
contrast	R13	Contrast value (20-100; default = 100)
fade	R13	Fade value (0-80; default = 0)

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.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.**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*.

UnderlayDefinition

class UnderlayDefinition(*GraphicEntity*)
Introduced in AutoCAD R13 (DXF version 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()*.

DXFAttr	Version	Description
filename	R13	Relative (to the DXF file) or absolute path to the image file as string
name	R13	defines what to display - pdf: page number; dgn: 'default'; dwf: ???

Mesh

class Mesh(*GraphicEntity*)
Introduced in AutoCAD R13 (DXF version AC1012), *dxftype* is MESH.

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

Mesh.**edit_data**()
Context manager various mesh data, returns *MeshData*.

See also:

Tutorial for Image and ImageDef

DXFAttr	Version	Description
version	R13	int
blend_crease	R13	0 = off, 1 = on
subdivision_levels	R13	int >= 0, 0 = no smoothing

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

Hatch

class Hatch
Introduced in AutoCAD R13 (DXF version AC1012), *dxftype* 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()*.

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 patten 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 = BYLAYER)
- **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.

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.

DXFAttr	Version	Description
pattern_name	R13	pattern name as string
solid_fill	R13	solid fill = 1, pattern fill = 0 (better use: *Hatch.set_solid_fill()*, *Hatch.set_pattern_fill()*)
associative	R13	1 for associative hatch else 0, associations not handled by ezdxf, you have to set the handles to the associated DXF entities by yourself.
hatch_style	R13	0 = normal; 1 = outer; 2 = ignore (search for AutoCAD help for more information)
pattern_type	R13	0 = user; 1 = predefined; 2 = custom; (???)
pattern_angle	R13	pattern angle in degrees (360 deg = circle)
pattern_scale	R13	as float
pattern_double	R13	1 = double else 0
n_seed_points	R13	count of seed points (better user: *Hatch.get_seed_points()*)

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**
external(1) or outermost(16) or default (0) - polyline(2) will be set by *ezdxf*

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**
external(1) or outermost(16) or default (0), 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 amd 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

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 *KeyError* 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):
Delete block *name*. Raises *KeyError* if block not exists.

BlockSection.delete_all_blocks():

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

Block Definition

class Block
Blocks are embedded into the *BlockLayout* object.

Block Reference

class Insert
A block reference with the possibility to append attributes (*Attrib*).

DXFAttr	Version	Description
layer	R12	layer name (str), default is 0
linetype	R12	linetype name or special name BYLAYER (str), default is BYLAYER
color	R12	dxf color index (int), 256 ... BYLAYER, default is 256
name	R12	block name (str)
insert	R12	insertion point as (2D/3D Point)
xscale	R12	scale factor for x direction (float)
yscale	R12	scale factor for y direction (float)
zscale	R12	scale factor for z direction (float)
rotation	R12	rotation angle in degrees (float)
row_count	R12	count of repeated insertions in row direction (int)
row_spacing	R12	distance between two insert points in row direction (float)
column_count	R12	count of repeated insertions in column direction (int)
column_spacing	R12	distance between two insert points in column direction (float)

Insert.**dxf**
DXF attributes namespace, read/write DXF attributes, like `object.dxf.layer = 'MyLayer'`

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.**attribs**()
Iterate over appended *Attrib* objects.

Insert.**has_attrib**(*tag*, *search_const=False*)
Returns *True* if an attrib *tag* exists else *False*, for *search_const* doc see *Insert.get_attrib()*.

Insert.**get_attrib**(*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_attrib_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_attrib()*.

Insert.**add_attrib**(*tag*, *text*, *insert=(0, 0)*, *attribs={}*)
Append an *Attrib* to the block reference. Returns an :class:Attrib object.

Example for appending an attribute to an INSERT entity with non standard alignment:

```
insert_entity.add_attrib("TAG", "example text").set_pos((3, 7), align='MIDDLE_CENTER')
```

Attribs

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.

DXFAttr	Version	Description
text	R12	the default text prompted by CAD programs (str)
insert	R12	first alignment point of text (2D/3D Point), relevant for the adjustments `LEFT`, `ALIGN` and `FIT`.
tag	R12	tag to identify the attribute (str)
align_point	R12	second alignment point of text (2D/3D Point), 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`)
height	R12	text height in drawing units (float), default is 1
rotation	R12	text rotation in degrees (float), default is 0
oblique	R12	text oblique angle (float), default is 0
style	R12	text style name (str), default is `STANDARD`
width	R12	width scale factor (float), default is 1
halign	R12	horizontal alignment flag (int), use *Attdef.set_pos()* and *Attdef.set_align()*
valign	R12	vertical alignment flag (int), use *Attdef.set_pos()* and *Attdef.set_align()*
text_generation_flag	R12	**text generation flags (int)** • 2 = text is backward (mirrored in X) • 4 = text is upside down (mirrored in Y)
prompt	R12	text prompted by CAD programs at placing a block reference containing this *Attdef*
field_length	R12	just relevant to CAD programs for validating user input

`Attdef.`**`dxf`**
DXF attributes namespace, read/write DXF attributes, like `object.dxf.layer = 'MyLayer'`

`Attdef.`**`is_invisibe`**
(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.`**`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()`*.

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.

DXFAttr	Version	Description
text	R12	the content text (str)
insert	R12	first alignment point of text (2D/3D Point), relevant for the adjustments `LEFT`, `ALIGN` and `FIT`.
tag	R12	tag to identify the attribute (str)
align_point	R12	second alignment point of text (2D/3D Point), 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`)
height	R12	text height in drawing units (float), default is 1
rotation	R12	text rotation in degrees (float), default is 0
oblique	R12	text oblique angle (float), default is 0
style	R12	text style name (str), default is `STANDARD`
width	R12	width scale factor (float), default is 1
halign	R12	horizontal alignment flag (int), use *`Attrib.set_pos()`* and *`Attrib.set_align()`*
valign	R12	vertical alignment flag (int), use *`Attrib.set_pos()`* and *`Attrib.set_align()`*
text_generation_flag	R12	**text generation flags (int)** • 2 = text is backward (mirrored in X) • 4 = text is upside down (mirrored in Y)

`Attrib.`**`dxf`**
DXF attributes namespace, read/write DXF attributes, like `object.dxf.layer = 'MyLayer'`

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.**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()*.

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 *KeyError* 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.

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 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 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.

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 a 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**(*dxfattrib=''*, *key=None*)

Returns a mapping of this result container, 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

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`*.

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 __ __ __ __ __ __ __ __ __ __ __ __ __ _
- DASHEDX2 ____ ____ ____ ____ ____ ____
- DASHED2 _ _ _ _ _ _ _ _ _ _ _ _ _ _
- PHANTOM _____ __ __ _____ __ __ _____
- PHANTOMX2 _____ ____ ____ _____
- PHANTOM2 ___ _ _ ___ _ _ ___ _ _ ___ _ _ ___
- DASHDOT __ . __ . __ . __ . __ . __ . __ . __
- DASHDOTX2 ____ . ____ . ____ . ____
- DASHDOT2 _ . _ . _ . _ . _ . _ . _ . _ . _
- DOT . . . . . .
- DOTX2 . . . . . .
- DOT2 . . . . . .
- DIVIDE __ . . __ . . __ . . __ . . __ . . __
- DIVIDEX2 ____ . . ____ . . ____ . . ____
- DIVIDE2 _ . _ . _ . _ . _ . _ . _ . _ . _

If using FIXED-TABLES, following predefined text styles are available:

- ARIAL
- ARIAL_NARROW
- ISOCPEUR

- TIMES

Tutorial

A simple example with different DXF entities:

```
from random import random
from ezdxf.r12writer import r12writer

with r12writer("quick_and_dirty_dxf_r12.dxf") as dxf:
    dxf.add_line((0, 0), (17, 23))
    dxf.add_circle((0, 0), radius=2)
    dxf.add_arc((0, 0), radius=3, start=0, end=175)
    dxf.add_solid([(0, 0), (1, 0), (0, 1), (1, 1)])
    dxf.add_point((1.5, 1.5))
    dxf.add_polyline([(5, 5), (7, 3), (7, 6)])  # 2d polyline
    dxf.add_polyline([(4, 3, 2), (8, 5, 0), (2, 4, 9)])  # 3d polyline
    dxf.add_text("test the text entity", align="MIDDLE_CENTER")
```

A simple example of writing really many entities in a short time:

```
from random import random
from ezdxf.r12writer import r12writer

MAX_X_COORD = 1000.0
MAX_Y_COORD = 1000.0
CIRCLE_COUNT = 1000000

with r12writer("many_circles.dxf") as dxf:
    for i in range(CIRCLE_COUNT):
        dxf.add_circle((MAX_X_COORD*random(), MAX_Y_COORD*random()), radius=2)
```

Show all available line types:

```
import ezdxf

LINETYPES = [
    'CONTINUOUS', 'CENTER', 'CENTERX2', 'CENTER2', 'DASHED', 'DASHEDX2', 'DASHED2',
↪'PHANTOM', 'PHANTOMX2',
    'PHANTOM2', 'DASHDOT', 'DASHDOTX2', 'DASHDOT2', 'DOT', 'DOTX2', 'DOT2', 'DIVIDE',
↪'DIVIDEX2', 'DIVIDE2',
]

with r12writer('r12_linetypes.dxf', fixed_tables=True) as dxf:
    for n, ltype in enumerate(LINETYPES):
        dxf.add_line((0, n), (10, n), linetype=ltype)
        dxf.add_text(ltype, (0, n+0.1), height=0.25, style='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 ot 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.

Howto

General preconditions:

```
import ezdxf
dwg = ezdxf.readfile("your_dxf_file.dxf")
modelspace = dwg.modelspace()
```

Get/Set block reference attributes

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

Iterate over all appended attributes:

```
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.attribs():
        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"
```

Reduce Memory Footprint

- compress binary data by *Drawing.compress_binary_data()*
- compress useless sections like *THUMBNAILIMAGE* by setting `ezdxf.options.compress_default_chunks = True`, *before* opening the DXF file.

Warning: Data compression costs time: *memory usage* vs *run time*

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

# 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.

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

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.

A360 do not open ezdxf files

AutoDesk web service A360 seems to more picky than the AutoCAD desktop applications, may be it helps to use the latest DXF version supported by ezdxf, which is `AC1027` in the year of writing this lines (2017).

DXF Internals

DXF File Encoding

The following facts are not in the DXF Standard. But this facts are established by the AutoCAD application.

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 files are saved with UTF-8 encoding, the header variable *$DWGCODEPAGE* is still in use, but I don't know if the setting has any meaning.

Encoding characters in the unicode schema `\U+nnnn` is still functional.

DXF File Structure

A Drawing Interchange File is simply an ASCII text file with a file type of .dxf and specially formatted text. The overall organization of a DXF file is as follows:

1. HEADER - General information about the drawing is found in this section of the DXF file. Each parameter has a variable name and an associated value.
2. CLASSES - This section holds the information for application-defined classes. This section was introduced with AC1015 and can usually be ignored.
3. TABLES - This section contains definitions of named items.

- Linetype table (LTYPE)
- Layer table (LAYER)
- Text Style table (STYLE)
- View table (VIEW)
- User Coordinate System table (UCS)
- Viewport configuration table (VPORT)
- Dimension Style table (DIMSTYLE)
- Application Identification table (APPID)

4. BLOCKS - This section contains Block Definition entities describing the entities that make up each Block in the drawing.
5. ENTITIES - This section contains the drawing entities, including any Block References.
6. OBJECTS - non-graphical objects
7. THUMBNAILIMAGE - This section contains a preview image of the DXF file, it is optional and can usually be ignored.
8. END OF FILE

By using *ezdxf* you don't have to know much about this details, but interested users can look at the original DXF Reference.

Minimal DXF Content

DXF R12

The DXF format R12 (AC1009) and prior requires just an ENTITIES section:

```
  0
SECTION
  2
ENTITIES
  0
ENDSEC
  0
EOF
```

DXF R13/14 and later

DXF version R13/14 and later need much more DXF content than DXF version 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 with at least an entry called `'*ACTIVE'`

- 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 `ACAD_GROUP`
- DICTONARY `ACAD_GROUP` can be empty

Minimal DXF to download: https://bitbucket.org/mozman/ezdxf/downloads/Minimal_DXF_AC1021.dxf

News

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 drawing 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()

Version 0.7.9 - 2017-01-31

- BUGFIX: lost data if model space and active layout are called *MODEL_SPACE and *PAPER_SPACE

Version 0.7.8 - 2017-01-22

- BUGFIX: HATCH accepts SplineEdges without defined fit points
- BUGFIX: fixed universal line ending problem in ZipReader()
- Moved repository to GitHub: https://github.com/mozman/ezdxf.git

Version 0.7.7 - 2016-10-22

- NEW: repairs malformed Leica Disto DXF R12 files, ezdxf saves a valid DXF R12 file.
- NEW: added Layout.unlink(entity) method: unlinks an entity from layout but does not delete entity from the drawing database.
- NEW: added Drawing.add_xref_def(filename, name) for adding external reference definitions
- CHANGE: renamed parameters for EdgePath.add_ellipse() - major_axis_vector -> major_axis; minor_axis_length -> ratio to be consistent to the ELLIPSE entity
- UPDATE: Entity.tags.new_xdata() and Entity.tags.set_xdata() accept tuples as tags, no import of DXFTag required
- UPDATE: EntityQuery to support both 'single' and "double" quoted strings - Harrison Katz <harrison@neadwerx.com>
- improved DXF R13/R14 compatibility

Version 0.7.6 - 2016-04-16

- NEW: r12writer.py - a fast and simple DXF R12 file/stream writer. Supports only LINE, CIRCLE, ARC, TEXT, POINT, SOLID, 3DFACE and POLYLINE. The module can be used without ezdxf.
- NEW: Get/Set extended data on DXF entity level, add and retrieve your own data to DXF entities
- NEW: Get/Set app data on DXF entity level (not important for high level users)
- NEW: Get/Set/Append/Remove reactors on DXF entity level (not important for high level users)
- CHANGE: using reactors in PdfDefinition for well defined UNDERLAY entities
- CHANGE: using reactors and IMAGEDEF_REACTOR for well defined IMAGE entities
- BUGFIX: default name=None in add_image_def()

Version 0.7.5 - 2016-04-03

- NEW: Drawing.acad_release property - AutoCAD release number for the drawing DXF version like 'R12' or 'R2000'
- NEW: support for PDFUNDERLAY, DWFUNDERLAY and DGNUNDERLAY entities
- BUGFIX: fixed broken layout setup in repair routine
- BUGFIX: support for utf-8 encoding on saving, DXF R2007 and later is saved with UTF-8 encoding
- CHANGE: Drawing.add_image_def(filename, size_in_pixel, name=None), renamed key to name and set name=None for auto-generated internal image name

- CHANGE: argument order of Layout.add_image(image_def, insert, size_in_units, rotation=0., dxfattribs=None)

Version 0.7.4 - 2016-03-13

- NEW: support for DXF entity IMAGE (work in progress)
- NEW: preserve leading file comments (tag code 999)
- NEW: writes saving and upgrading comments when saving DXF files; avoid this behavior by setting options.store_comments = False
- NEW: ezdxf.new() accepts the AutoCAD release name as DXF version string e.g. ezdxf.new('R12') or R2000, R2004, R2007, ...
- NEW: integrated acadctb.py module from my dxfwrite package to read/write AutoCAD .ctb config files; no docs so far
- CHANGE: renamed Drawing.groups.add() to new() for consistent name schema for adding new items to tables (public interface)
- CHANGE: renamed Drawing.<tablename>.create() to new() for consistent name schema for adding new items to tables, this applies to all tables: layers, styles, dimstyles, appids, views, viewports, ucs, block_records. (public interface)
- CHANGE: renamed Layouts.create() to new() for consistent name schema for adding new items to tables (internal interface)
- CHANGE: renamed Drawing.create_layout() to new_layout() for consistent name schema for adding new items (public interface)
- CHANGE: renamed factory method <layout>.add_3Dface() to add_3dface()
- REMOVED: logging and debugging options
- BUGFIX: fixed attribute definition for align_point in DXF entity ATTRIB (AC1015 and newer)
- Cleanup DXF template files AC1015 - AC1027, file size goes down from >60kb to ~20kb

Version 0.7.3 - 2016-03-06

- Quick bugfix release, because ezdxf 0.7.2 can damage DXF R12 files when saving!!!
- NEW: improved DXF R13/R14 compatibility
- BUGFIX: create CLASSES section only for DXF versions newer than R12 (AC1009)
- TEST: converted a bunch of R8 (AC1003) files to R12 (AC1009), AutoCAD didn't complain
- TEST: converted a bunch of R13 (AC1012) files to R2000 (AC1015), AutoCAD didn't complain
- TEST: converted a bunch of R14 (AC1014) files to R2000 (AC1015), AutoCAD didn't complain

Version 0.7.2 - 2016-03-05

- NEW: reads DXF R13/R14 and saves content as R2000 (AC1015) - experimental feature, because of the lack of test data
- NEW: added support for common DXF attribute line weight
- NEW: POLYLINE, POLYMESH - added properties is_closed, is_m_closed, is_n_closed
- BUGFIX: MeshData.optimize() - corrected wrong vertex optimization
- BUGFIX: can open DXF files without existing layout management table
- BUGFIX: restore module structure ezdxf.const

Version 0.7.1 - 2016-02-21

- Supported/Tested Python versions: CPython 2.7, 3.4, 3.5, pypy 4.0.1 and pypy3 2.4.0
- NEW: read legacy DXF versions older than AC1009 (DXF R12) and saves it as DXF version AC1009.
- NEW: added methods is_frozen(), freeze(), thaw() to class Layer()
- NEW: full support for DXF entity ELLIPSE (added add_ellipse() method)
- NEW: MESH data editor - implemented add_face(vertices), add_edge(vertices), optimize(precision=6) methods
- BUGFIX: creating entities on layouts works
- BUGFIX: entity ATTRIB - fixed halign attribute definition
- CHANGE: POLYLINE (POLYFACE, POLYMESH) - on layer change also change layer of associated VERTEX entities

Version 0.7.0 - 2015-11-26

- Supported Python versions: CPython 2.7, 3.4, pypy 2.6.1 and pypy3 2.4.0
- NEW: support for DXF entity HATCH (solid fill, gradient fill and pattern fill), pattern fill with background color supported
- NEW: support for DXF entity GROUP
- NEW: VIEWPORT entity, but creating new viewports does not work as expected - just for reading purpose.
- NEW: support for new common DXF attributes in AC1018 (AutoCAD 2004): true_color, color_name, transparency
- NEW: support for new common DXF attributes in AC1021 (AutoCAD 2007): shadow_mode
- NEW: extended custom vars interface
- NEW: dxf2html - added support for custom properties in the header section
- NEW: query() supports case insensitive attribute queries by appending an 'i' to the query string, e.g. '*[layer=="construction"]i'
- NEW: Drawing.cleanup() - call before saving the drawing but only if necessary, the process could take a while.
- BUGFIX: query parser couldn't handle attribute names containing '_'
- CHANGE: renamed dxf2html to pp (pretty printer), usage: py -m ezdxf.pp yourfile.dxf (generates yourfile.html in the same folder)
- CHANGE: cleanup file structure

CHAPTER 3

Indices and tables

- genindex
- search

Index

Symbols

__contains__() (BlocksSection method), 64
__contains__() (DXFGroup method), 69
__contains__() (DXFGroupTable method), 70
__contains__() (Layout method), 33
__contains__() (Table method), 29
__getitem__() (BlocksSection method), 64
__getitem__() (Face method), 46
__getitem__() (HeaderSection method), 28
__getitem__() (LWPolyline method), 48
__getitem__() (MeshVertexCache method), 44
__getitem__() (Polyline method), 42
__iadd__() (MTextData method), 49
__init__() (EntityQuery method), 74
__init__() (Importer method), 71
__init__() (R12FastStreamWriter method), 77
__iter__() (BlocksSection method), 64
__iter__() (CustomVars method), 28
__iter__() (DXFGroup method), 69
__iter__() (DXFGroupTable method), 70
__iter__() (Face method), 45
__iter__() (Layout method), 33
__iter__() (Table method), 29
__len__() (CustomVars method), 28
__len__() (DXFGroup method), 69
__len__() (DXFGroupTable method), 70
__len__() (Face method), 45
__len__() (LWPolyline method), 48
__len__() (Polyline method), 42
__len__() (Table method), 29
__setitem__() (HeaderSection method), 28
__setitem__() (MeshVertexCache method), 44
__str__() (ModelerGeometryData method), 53
3DFace (built-in class), 46
3DSolid (built-in class), 53

A

acad_version (Drawing attribute), 25
add_3dface() (Layout method), 33
add_3dface() (R12FastStreamWriter method), 78
add_3dsolid() (Layout method), 34
add_arc() (EdgePath method), 61
add_arc() (Layout method), 33
add_arc() (R12FastStreamWriter method), 77
add_attdef() (BlockLayout method), 35
add_attrib() (Insert method), 66
add_attrib() (Layout method), 34
add_auto_blockref() (Layout method), 33
add_blockref() (Layout method), 33
add_body() (Layout method), 34
add_circle() (Layout method), 33
add_circle() (R12FastStreamWriter method), 77
add_edge() (MeshData method), 57
add_edge_path() (BoundaryPathData method), 60
add_ellipse() (EdgePath method), 61
add_ellipse() (Layout method), 33
add_entity() (Layout method), 35
add_face() (MeshData method), 57
add_hatch() (Layout method), 34
add_image() (Layout method), 34
add_image_def() (Drawing method), 27
add_line() (EdgePath method), 61
add_line() (Layout method), 33
add_line() (PatternData method), 63
add_line() (R12FastStreamWriter method), 77
add_lwpolyline() (Layout method), 34
add_mtext() (Layout method), 34
add_point() (Layout method), 33
add_point() (R12FastStreamWriter method), 77
add_polyface() (Layout method), 34
add_polyline() (R12FastStreamWriter method), 78
add_polyline2d() (Layout method), 34
add_polyline3d() (Layout method), 34
add_polyline_path() (BoundaryPathData method), 60
add_polymesh() (Layout method), 34
add_ray() (Layout method), 34
add_region() (Layout method), 34
add_shape() (Layout method), 34
add_solid() (Layout method), 33

add_solid() (R12FastStreamWriter method), 78
add_spline() (EdgePath method), 61
add_spline() (Layout method), 34
add_text() (Layout method), 33
add_text() (R12FastStreamWriter method), 78
add_trace() (Layout method), 33
add_underlay() (Layout method), 35
add_underlay_def() (Drawing method), 27
add_xline() (Layout method), 34
add_xref_def() (Drawing method), 27
adjust_for_background (Underlay attribute), 55
angle (PatternDefinitionLine attribute), 63
append() (CustomVars method), 29
append() (MTextData method), 49
append_face() (Polyface method), 45
append_faces() (Polyface method), 45
append_points() (LWPolyline method), 48
append_vertices() (Polyline method), 43
AppID (built-in class), 32
appids (Drawing attribute), 26
Arc (built-in class), 38
ArcEdge (built-in class), 62
Attdef (built-in class), 66
attdefs() (BlockLayout method), 35
Attrib (built-in class), 68
attribs() (Insert method), 65

B

base_point (PatternDefinitionLine attribute), 63
bgcolor (Hatch attribute), 57
block (BlockLayout attribute), 35
Block (built-in class), 65
BlockLayout (built-in class), 35
BlockRecord (built-in class), 32
blocks (Drawing attribute), 25
BlocksSection (built-in class), 64
Body (built-in class), 52
BoundaryPathData (built-in class), 59

C

center (ArcEdge attribute), 62
centered (GradientData attribute), 64
Circle (built-in class), 38
cleanup() (Drawing method), 27
cleanup() (DXFGroupTable method), 70
clear() (BoundaryPathData method), 60
clear() (CustomVars method), 28
clear() (DXFGroup method), 70
clear() (DXFGroupTable method), 70
clear() (EdgePath method), 61
clear() (PatternData method), 63
clear() (PolylinePath method), 60
clipping (Underlay attribute), 55
close() (Polyline method), 42
close() (R12FastStreamWriter method), 77
closed (LWPolyline attribute), 47
closed (Spline attribute), 51
color1 (GradientData attribute), 64
color2 (GradientData attribute), 64
compress_binary_data (ezdxf.options attribute), 24
compress_binary_data() (Drawing method), 28
compress_default_chunks (ezdxf.options attribute), 24
control_points (SplineData attribute), 52
control_points (SplineEdge attribute), 63
custom_vars (HeaderSection attribute), 28
CustomVars (built-in class), 28

D

dash_length_items (PatternDefinitionLine attribute), 63
degree (SplineEdge attribute), 62
del_dxf_attrib() (GraphicEntity method), 37
delete() (DXFGroupTable method), 70
delete_all_entities() (Layout method), 35
delete_entity() (Layout method), 35
delete_layout() (Drawing method), 26
delete_vertices() (Polyline method), 43
DimStyle (built-in class), 31
dimstyles (Drawing attribute), 26
discard_points() (LWPolyline method), 48
Drawing (built-in class), 24
drawing (GraphicEntity attribute), 36
dxf (AppID attribute), 32
dxf (Attdef attribute), 67
dxf (Attrib attribute), 68
dxf (BlockRecord attribute), 32
dxf (DimStyle attribute), 31
dxf (GraphicEntity attribute), 36
dxf (Insert attribute), 65
dxf (Layer attribute), 30
dxf (Linetype attribute), 31
dxf (Style attribute), 30
dxf (UCS attribute), 32
dxf (View attribute), 32
dxf (Viewport attribute), 31
dxf_attrib_exists() (GraphicEntity method), 37
dxffactory (Drawing attribute), 25
dxffactory (GraphicEntity attribute), 36
DXFGroup (built-in class), 69
DXFGroupTable (built-in class), 70
dxftype (GraphicEntity attribute), 36
dxfversion (Drawing attribute), 25

E

edge_crease_values (MeshData attribute), 57
EdgePath (built-in class), 60
edges (EdgePath attribute), 60
edges (MeshData attribute), 57
edit_boundary() (Hatch method), 57

edit_data() (3DSolid method), 53
edit_data() (Body method), 53
edit_data() (DXFGroup method), 69
edit_data() (Mesh method), 56
edit_data() (MText method), 49
edit_data() (Region method), 53
edit_data() (Spline method), 52
edit_gradient() (Hatch method), 58
edit_pattern() (Hatch method), 57
Ellipse (built-in class), 39
EllipseEdge (built-in class), 62
encoding (Drawing attribute), 25
end (LineEdge attribute), 62
end_angle (ArcEdge attribute), 62
end_angle (EllipseEdge attribute), 62
end_tangent (SplineEdge attribute), 63
entities (Drawing attribute), 25
EntityQuery (built-in class), 74
extend() (DXFGroup method), 70
extend() (EntityQuery method), 74
ezdxf.new() (built-in function), 23
ezdxf.read() (built-in function), 23
ezdxf.readfile() (built-in function), 23

F

Face (built-in class), 45
face_record (Face attribute), 45
faces (MeshData attribute), 56
faces() (Polyface method), 45
filename (Drawing attribute), 25
fit_points (SplineData attribute), 52
fit_points (SplineEdge attribute), 63
freeze() (Layer method), 30

G

get() (BlocksSection method), 64
get() (CustomVars method), 29
get() (DXFGroupTable method), 70
get() (Table method), 29
get_acis_data() (3DSolid method), 53
get_acis_data() (Body method), 53
get_acis_data() (Region method), 53
get_align() (Attdef method), 68
get_align() (Attrib method), 69
get_align() (Text method), 41
get_attdef() (BlockLayout method), 36
get_attdef_text() (BlockLayout method), 36
get_attrib() (Insert method), 65
get_attrib_text() (Insert method), 65
get_boundary() (Image method), 54
get_boundary() (Underlay method), 56
get_color() (Layer method), 30
get_control_points() (Spline method), 52
get_dxf_attrib() (GraphicEntity method), 36
get_dxf_entity() (Drawing method), 28
get_fit_points() (Spline method), 52
get_gradient() (Hatch method), 58
get_image_def() (Image method), 54
get_knot_values() (Spline method), 52
get_mesh_vertex() (Polymesh method), 44
get_mesh_vertex_cache() (Polymesh method), 44
get_mode() (Polyline method), 42
get_name() (DXFGroup method), 69
get_points() (LWPolyline method), 47
get_pos() (Attdef method), 68
get_pos() (Attrib method), 69
get_pos() (Text method), 41
get_rotation() (MText method), 49
get_rstrip_points() (LWPolyline method), 47
get_seed_points() (Hatch method), 59
get_text() (MText method), 49
get_underlay_def() (Underlay method), 56
get_weights() (Spline method), 52
GradientData (built-in class), 64
GraphicEntity (built-in class), 36
grid() (Insert method), 65
groupby() (EntityQuery method), 74
groupby() (Layout method), 33
groups (Drawing attribute), 25
groups() (DXFGroupTable method), 70

H

handle (GraphicEntity attribute), 36
handles() (DXFGroup method), 69
has_attdef() (BlockLayout method), 36
has_attrib() (Insert method), 65
has_gradient_fill (Hatch attribute), 57
has_pattern_fill (Hatch attribute), 57
has_solid_fill (Hatch attribute), 57
has_tag() (CustomVars method), 29
Hatch (built-in class), 57
header (Drawing attribute), 25
HeaderSection (built-in class), 28

I

Image (built-in class), 54
ImageDef (built-in class), 55
import_all() (Importer method), 72
import_blocks() (Importer method), 72
import_modelspace_entities() (Importer method), 72
import_table() (Importer method), 71
import_tables() (Importer method), 71
Importer (built-in class), 71
indexed_faces() (Polyface method), 45
indices (Face attribute), 45
Insert (built-in class), 65
insert_vertices() (Polyline method), 43
is_2d_polyline (Polyline attribute), 42

is_3d_polyline (Polyline attribute), 42
is_binary_data_compressed (Drawing attribute), 26
is_closed (Polyline attribute), 42
is_closed (PolylinePath attribute), 60
is_compatible() (Importer method), 72
is_const (Attdef attribute), 67
is_const (Attrib attribute), 69
is_counter_clockwise (ArcEdge attribute), 62
is_counter_clockwise (EllipseEdge attribute), 62
is_edge_visible() (Face method), 46
is_frozen() (Layer method), 30
is_invisibe (Attdef attribute), 67
is_invisibe (Attrib attribute), 68
is_locked() (Layer method), 30
is_m_closed (Polyline attribute), 42
is_n_closed (Polyline attribute), 42
is_off() (Layer method), 30
is_on() (Layer method), 30
is_poly_face_mesh (Polyline attribute), 42
is_polygon_mesh (Polyline attribute), 42
is_preset (Attdef attribute), 67
is_preset (Attrib attribute), 69
is_verify (Attdef attribute), 67
is_verify (Attrib attribute), 69

K

knot_values (SplineData attribute), 52
knot_values (SplineEdge attribute), 63

L

Layer (built-in class), 30
layers (Drawing attribute), 26
Layout (built-in class), 32
layout() (Drawing method), 26
layout_names() (Drawing method), 26
Line (built-in class), 38
LineEdge (built-in class), 62
lines (PatternData attribute), 63
Linetype (built-in class), 31
linetypes (Drawing attribute), 26
lock() (Layer method), 30
LWPolyline (built-in class), 47

M

m_close() (Polyline method), 42
major_axis_vector (EllipseEdge attribute), 62
Mesh (built-in class), 56
MeshData (built-in class), 56
MeshVertexCache (built-in class), 44
minor_axis_length (EllipseEdge attribute), 62
Modelspace (built-in class), 35
modelspace() (Drawing method), 26
monochrome (Underlay attribute), 55
MText (built-in class), 48
MTextData (built-in class), 49

N

n_close() (Polyline method), 42
name (BlockLayout attribute), 35
new() (BlocksSection method), 64
new() (DXFGroupTable method), 70
new() (ezdxf.query method), 74
new() (Table method), 29
new_anonymous_block() (BlocksSection method), 64
new_layout() (Drawing method), 26
new_line() (PatternData method), 63

O

off() (Layer method), 30
offset (PatternDefinitionLine. attribute), 63
on (Underlay attribute), 55
on() (Layer method), 30
one_color (GradientData attribute), 64
optimize() (MeshData method), 57
optimize() (Polyface method), 45

P

Paperspace (built-in class), 35
path_type_flags (EdgePath attribute), 60
path_type_flags (PolylinePath attribute), 60
paths (BoundaryPathData attribute), 59
PatternData (built-in class), 63
PatternDefinitionLine (built-in class), 63
periodic (SplineEdge attribute), 63
place() (Insert method), 65
Point (built-in class), 38
points() (Face method), 46
points() (LWPolyline method), 47
points() (Polyline method), 43
Polyface (built-in class), 45
Polyline (built-in class), 41
PolylinePath (built-in class), 60
Polymesh (built-in class), 44
properties (CustomVars attribute), 28

Q

query() (EntityQuery method), 74
query() (Layout method), 33

R

R12FastStreamWriter (built-in class), 76
r12writer() (built-in function), 76
radius (ArcEdge attribute), 62
radius (EllipseEdge attribute), 62
rational (SplineEdge attribute), 63
Ray (built-in class), 50
Region (built-in class), 53

remove() (CustomVars method), 29
remove() (EntityQuery method), 74
remove() (Table method), 29
remove_invalid_handles() (DXFGroup method), 70
rename_block() (BlocksSection method), 64
replace() (CustomVars method), 29
reset_boundary() (Image method), 54
reset_boundary() (Underlay method), 56
rgb (GraphicEntity attribute), 36
rotation (GradientData attribute), 64
rstrip_points() (LWPolyline method), 48

S

save() (Drawing method), 27
saveas() (Drawing method), 27
scale (Underlay attribute), 56
sections (Drawing attribute), 25
set_acis_data() (3DSolid method), 53
set_acis_data() (Body method), 53
set_acis_data() (Region method), 53
set_align() (Attdef method), 68
set_align() (Attrib method), 69
set_align() (Text method), 41
set_boundary() (Image method), 54
set_boundary() (Underlay method), 56
set_color() (Layer method), 30
set_color() (MTextData method), 50
set_control_points() (Spline method), 52
set_data() (DXFGroup method), 70
set_dxf_attrib() (GraphicEntity method), 36
set_fit_points() (Spline method), 52
set_font() (MTextData method), 50
set_gradient() (Hatch method), 58
set_knot_values() (Spline method), 52
set_location() (MText method), 49
set_mesh_vertex() (Polymesh method), 44
set_pattern_definition() (Hatch method), 57
set_pattern_fill() (Hatch method), 59
set_points() (LWPolyline method), 47
set_pos() (Attdef method), 68
set_pos() (Attrib method), 69
set_pos() (Text method), 40
set_rotation() (MText method), 49
set_seed_points() (Hatch method), 59
set_solid_fill() (Hatch method), 58
set_text() (MText method), 49
set_vertices() (PolylinePath method), 60
set_weights() (Spline method), 52
Shape (built-in class), 50
Solid (built-in class), 46
source_boundary_objects (EdgePath attribute), 61
source_boundary_objects (PolylinePath attribute), 60
Spline (built-in class), 51
SplineData (built-in class), 52
SplineEdge (built-in class), 62
start (LineEdge attribute), 62
start_angle (ArcEdge attribute), 62
start_angle (EllipseEdge attribute), 62
start_tangent (SplineEdge attribute), 63
store_comments (ezdxf.options attribute), 24
Style (built-in class), 30
styles (Drawing attribute), 26
supported_dxf_attrib() (GraphicEntity method), 37

T

Table (built-in class), 29
templatedir (ezdxf.options attribute), 24
Text (built-in class), 39
text (MTextData attribute), 49
text_lines (ModelerGeometryData attribute), 53
thaw() (Layer method), 30
tint (GradientData attribute), 64
Trace (built-in class), 46
transparency (GraphicEntity attribute), 36

U

UCS (built-in class), 32
ucs (Drawing attribute), 26
Underlay (built-in class), 55
UnderlayDefinition (built-in class), 56
unlink_entity() (Layout method), 35
unlock() (Layer method), 30

V

valid_dxf_attrib_names() (GraphicEntity method), 37
Vertex (built-in class), 43
vertices (Face attribute), 45
vertices (MeshData attribute), 56
vertices (MeshVertexCache attribute), 44
vertices (PolylinePath attribute), 60
vertices() (Polyline method), 43
View (built-in class), 32
Viewport (built-in class), 31
viewports (Drawing attribute), 26
views (Drawing attribute), 26

W

weights (SplineData attribute), 52
weights (SplineEdge attribute), 63
write() (Drawing method), 27

X

XLine (built-in class), 51