ezdxf Documentation

Release 0.8.7

Manfred Moitzi

Mar 04, 2018

Contents

Welcome! This is the documentation for ezdxf 0.8.7, last updated Mar 04, 2018.

- *ezdxf* is a Python package to create new DXF files and read/modify/write existing DXF files
- the intended audience are developers
- requires Python 2.7 or later, runs on CPython and pypy, maybe on IronPython and Jython
- OS independent
- additional required packages: pyparsing
- MIT-License
- read/write/new support for DXF versions: R12, R2000, R2004, R2007, R2010, R2013 and R2018
- additional read support for DXF versions R13/R14 (upgraded to R2000)
- additional read support for older DXF versions than R12 (upgraded to R12)
- preserves third-party DXF content
- additional *Fast DXF R12 File/Stream Writer*, that creates just an ENTITIES section with support for the basic DXF entities

CHAPTER 1

Website

https://ezdxf.mozman.at/

CHAPTER 2

Documentation

Documentation of development version at https://ezdxf.mozman.at/docs

Documentation of latest release at http://ezdxf.readthedocs.io/

Source Code: http://github.com/mozman/ezdxf.git

Issue Tracker at GitHub: http://github.com/mozman/ezdxf/issues

CHAPTER 3

Questions and Feedback at Google Groups

https://groups.google.com/d/forum/python-ezdxf

CHAPTER 4

Contents

4.1 Introduction

4.1.1 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.

4.1.2 What ezdxf is NOT

- *ezdxf* is not a DXF converter: *ezdxf* can not convert between different DXF versions, if you are looking for an appropriate program, use *DWG TrueView* from Autodesk, but the latest version can only convert to the DWG format, for converting between DXF versions you need at least AutoCAD LT.
- *ezdxf* is not a CAD file format converter: *ezdxf* can not convert DXF files to **ANY** other format, like SVG, PDF or DWG
- *ezdxf* is not a DXF renderer (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.

4.1.3 Supported Python Versions

ezdxf requires at least Python 2.7 and it's Python 3 compatible. I run unit tests with the latest stable CPython 3 version and the latest stable release of pypy during development. *ezdxf* is written in pure Python and requires only *pyparser* as additional library beside the Python Standard Library, hence it should run with IronPython and Jython also. *pytest*

is required to run the provided unit and integration tests. Data to run the stress and audit test can not be provided, because I don't have the publishing rights for this DXF files.

4.1.4 Supported Operating Systems

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

4.1.5 Supported DXF Versions

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

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

4.1.6 Embedded DXF Information of 3rd Party Applications

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

4.1.7 License

ezdxf is licensed under the very liberal MIT-License.

4.2 Tutorials

4.2.1 Tutorial for Getting Data from DXF Files

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

At first load the drawing:

```
import ezdxf

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

See also:

Drawing Management

Layouts

I use the term layout as synonym for an arbitrary entity space which can contain any DXF 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`

4.2.2 Tutorial for Creating Simple DXF Drawings

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

See also:

Fast DXF R12 File/Stream Writer

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

```
import ezdxf

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

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

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

See also:

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

4.2.3 Tutorial for Layers

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

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

First you have to create layers, assigning them names and properties such as color and linetype. Then you can assign those layers to other drawing entities. To assign a layer just use its name as string. It is not recommend but it is possible to use layers without a layer definition, just use the layer name without a definition, the layer has the default linetype *Continuous* and the default color is *1*.

Create a new layer definition:

```
import ezdxf

dwg = ezdxf.new()
msp = modelspace()
dwg.layers.new(name='MyLines', dxfattribs={'linetype': 'DASHED', 'color': 7})
```

The advantage of assigning a linetype and a color to a layer is that entities on this layer can inherit this properties by using `BYLAYER` as linetype string 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*.

4.2.4 Tutorial for Blocks

What are Blocks?

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

Create a Block

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

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


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

# Create a new drawing in the DXF format of AutoCAD 2010
dwg = ezdxf.new('ac1024')

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

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

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)
```

4.2.5 Tutorial for LWPolyline

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

Create a simple polyline:

```
import ezdxf

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

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

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

Append points to a polyline:

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

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

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

Getting points always returns a 5-tuple (x, y, start_width, 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.

4.2.6 Tutorial for Text

TEXT - just one line

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

```
import ezdxf

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

msp = dwg.modelspace()

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

4.2.7 Tutorial for MText

coming soon ...

4.2.8 Tutorial for Spline

Create a simple spline:

```
import ezdxf

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

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

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

Add a fit point to a spline:

```
import ezdxf

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

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

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

    points = data.fit_points[:-1]  # pitfall: this creates a new list without a␣
↪connection to the spline object
```

```
    points.append((3000, 3000, 0))  # has no effect for the spline object

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

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

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

You can set additional *control points*, but if they do not fit the auto-generated AutoCAD values, they will be ignored and don't mess around with *knot values*.

Solve problems of incorrect values after editing an AutoCAD generated file:

```
import ezdxf

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

msp = dwg.modelspace()
spline = msp.query('SPLINE')[0]  # take the first spline
with spline.edit_data() as data:  # context 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())
```

4.2.9 Tutorial for Polyface

coming soon ...

4.2.10 Tutorial for Mesh

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

```
import ezdxf


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

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

dwg = ezdxf.new('AC1015')  # mesh requires the DXF 2000 or newer format
msp = dwg.modelspace()
mesh = msp.add_mesh()
mesh.dxf.subdivision_levels = 0  # do not subdivide cube, 0 is the default value
with mesh.edit_data() as mesh_data:
    mesh_data.vertices = cube_vertices
    mesh_data.faces = cube_faces

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

Create a cube mesh by method calls:

```
import ezdxf


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

```
    (0, 1, 1),
]

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

with mesh.edit_data() as mesh_data:
    mesh_data.add_face([p[0], p[1], p[2], p[3]])
    mesh_data.add_face([p[4], p[5], p[6], p[7]])
    mesh_data.add_face([p[0], p[1], p[5], p[4]])
    mesh_data.add_face([p[1], p[2], p[6], p[5]])
    mesh_data.add_face([p[3], p[2], p[6], p[7]])
    mesh_data.add_face([p[0], p[3], p[7], p[4]])
    mesh_data.optimize()  # optional, minimizes vertex count

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

4.2.11 Tutorial for Hatch

Create hatches with one boundary path

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

```
import ezdxf

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

hatch = msp.add_hatch(color=2)  # by default a solid fill hatch with fill color=7␣
↪(white/black)
with hatch.edit_boundary() as boundary:  # edit boundary path (context manager)
    # every boundary path is always a 2D element
    # vertex format for the polyline path is: (x, y[, bulge])
    # there are no bulge values in this example
    boundary.add_polyline_path([(0, 0), (10, 0), (10, 10), (0, 10)], is_closed=1)

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

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

```
import ezdxf

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

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

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

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

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

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

```
import ezdxf

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

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

hatch = msp.add_hatch(color=2)  # by default a solid fill hatch with fill color=7␣
↪(white/black)
with hatch.edit_boundary() as boundary:  # edit boundary path (context manager)
    # every boundary path is always a 2D element
    edge_path = boundary.add_edge_path()
    # each edge path can contain line arc, ellipse and spline elements
    # important: major axis >= minor axis (ratio <= 1.)
    edge_path.add_ellipse((0, 0), major_axis=(0, 10), ratio=0.5)

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

Create hatches with multiple boundary paths (islands)

TODO

Create hatches with with pattern fill

TODO

Create hatches with gradient fill

TODO

4.2.12 Tutorial for Hatch Pattern Definition

TODO

4.2.13 Tutorial for Image and ImageDef

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

```
import ezdxf


dwg = ezdxf.new('AC1015')  # image requires the DXF R2000 format or later
my_image_def = dwg.add_image_def(filename='mycat.jpg', size_in_pixel=(640, 360))
# The IMAGEDEF entity is like a block definition, it just defines the image

msp = dwg.modelspace()
# add first image
msp.add_image(insert=(2, 1), size_in_units=(6.4, 3.6), image_def=my_image_def,␣
↪rotation=0)
# The IMAGE entity is like the INSERT entity, it creates an image reference,
# and there can be multiple references to the same picture in a drawing.

msp.add_image(insert=(4, 5), size_in_units=(3.2, 1.8), image_def=my_image_def,␣
↪rotation=30)

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

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

4.2.14 Tutorial for Underlay and UnderlayDefinition

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

```
import ezdxf


dwg = ezdxf.new('AC1015')  # underlay requires the DXF R2000 format or later
my_underlay_def = dwg.add_underlay_def(filename='my_underlay.pdf', name='1')
# The (PDF)DEFINITION entity is like a block definition, it just defines the underlay
# 'name' is misleading, because it defines the page/sheet to be displayed
# PDF: name is the page number to display
# DGN: name='default' ???
# DWF: ????

msp = dwg.modelspace()
# add first underlay
msp.add_underlay(my_underlay_def, insert=(2, 1, 0), scale=0.05)
# The (PDF)UNDERLAY entity is like the INSERT entity, it creates an underlay␣
↪reference,
# and there can be multiple references to the same underlay in a drawing.

msp.add_underlay(my_underlay_def, insert=(4, 5, 0), scale=.5, rotation=30)

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

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

4.2.15 Tutorial for Linetypes

Simple line type example:

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

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

total_pattern_length Sum of all linetype elements (absolute vaues)

elem if elem > 0 it is a line, if elem < 0 it is gap, if elem == 0.0 it is a dot

Create a new linetype definition:

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

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

my_line_types = [
    ("DOTTED", "Dotted .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .", [0.2, 0.0, -0.
↪2]),
    ("DOTTEDX2", "Dotted (2x) .    .    .    .    .    .    .    . ", [0.4, 0.0, -0.
↪4]),
    ("DOTTED2", "Dotted (.5) . . . . . . . . . . . . . . . . . . . . ", [0.1, 0.0, -0.
↪1]),
]
for name, desc, pattern in my_line_types:
    if name not in dwg.linetypes:
        dwg.linetypes.new(name=name, dxfattribs={'description': desc, 'pattern':␣
↪pattern})
```

Setup some predefined linetypes:

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

Check Available Linetypes

The linetypes object supports some standard Python protocols:

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

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

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

Removing Linetypes

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

You can delete a linetype:

```
dwg.layers.remove('DASHED')
```

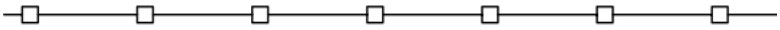
This just deletes the linetype definition, all DXF entity with the DXF attribute linetype set to `DASHED` still refers to linetype `DASHED` and AutoCAD will not open DXF files with undefined line types.

4.2.16 Tutorial for Complex Linetypes

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

Complex line type example with text:

Complex line type example with shapes:



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

Example for complex line type TEXT:

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

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

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

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

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

The parameters are case insensitive.] ends the complex part of the line pattern.

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

Example for complex line type SHAPE:

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

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

4.3 Reference

The DXF Reference is online available at Autodesk.

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

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

4.3.1 Drawing

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

Drawing Management

Create New Drawings

ezdxf.**new**(*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
AC1032	AutoCAD R2018

Open Drawings

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

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

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

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

Hint: Try option *legacy_mode=True* if error "Missing required y coordinate near line: ..." occurs.

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

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

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

Save Drawings

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

Global Options

Global options stored in `ezdxf.options`

`ezdxf.options.`**`compress_binary_data`**

If you don't need access to binary data of DXF entities, you can compress them in memory for a lower memory footprint, set the global `ezdxf.options.compress_binary_data = True` to compress 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.`**`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.

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.

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.

4.3.2 Tables

Table Class

Generic Table Class

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

`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.**has_entry**(*name*)
True if table contains a table-entry *name*.

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

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

Style Table Class

class StyleTable(*Table*)

StyleTable.**get_shx**(*name*)
Get existing shx entry, or create a new entry.

StyleTable.**find_shx**(*name*)
Find .shx shape file table entry, by a case insensitive search. A .shx shape file table entry has no name, so you have to search by the font attribute.

Viewport Table Class

class ViewportTable(*Table*)
The viewport table stores the model space viewport configurations. A viewport configuration is a tiled view of multiple viewports or just one viewport. In contrast to other tables the viewport table can have multiple entries with the same name, because all viewport entries of a multi-viewport configuration are having the same name - the viewport configuration name.

The name of the actual displayed viewport configuration is `*ACTIVE`.

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

ViewportTable.delete_config(name):
Delete all `Viewport` objects of the multi-viewport configuration *name*.

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

See also:

DXF Internals: *LTYPE Table*

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

VPort

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

See also:

DXF Internals: *VPORT Configuration Table*

class VPort
Defines a viewport to the model space.

VPort.**dxf**
The DXF attributes namespace.

TODO DXFAttr for the Viewport class

View

The View table stores named views of the model or paper space layouts. This stored views makes parts of the drawing or some view points of the model in a CAD applications more accessible. This views have no influence to the drawing content or to the generated output by exporting PDFs or plotting on paper sheets, they are just for the convenience of CAD application users.

See also:

DXF Internals: *VIEW Table*

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

4.3.3 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

Paper Space Layout Setup

Layout.**page_setup**(*size=(297, 210)*, *margins=(10, 15, 10, 15)*, *units='mm'*, *offset=(0, 0)*, *rotation=0*, *scale=16*, *name='ezdxf'*, *device='DWG to PDF.pc3'*)

Setup plot settings and paper size and reset viewports. All parameters in given *units* (mm or inch). DXF R12 not supported yet.

Parameters

- **size** – paper size as (width, height) tuple
- **margins** – (top, right, bottom, left) hint: clockwise
- **units** – 'mm' or 'inch'
- **offset** – plot origin offset as (x, y) tuple
- **rotation** – 0=no rotation, 1=90deg count-clockwise, 2=upside-down, 3=90deg clockwise
- **scale** – int 0-32 = standard scale type or tuple(numerator, denominator) e.g. (1, 50) for 1:50
- **name** – paper name prefix '{name}_({width}_x_{height}_{unit})'
- **device** – device .pc3 configuration file or system printer name

Layout.**reset_viewports**()

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

Layout.**reset_extends**()

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

Layout.**reset_paper_limits**()

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

Layout.**get_paper_limits**()

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

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

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

Set plot type:

- 0 = last screen display

- 1 = drawing extents
- 2 = drawing limits
- 3 = view specific (defined by Layout.dxf.plot_view_name)
- 4 = window specific (defined by Layout.set_plot_window_limits())
- 5 = layout information (default)

Layout.**set_plot_style**(*name='ezdxf.ctb'*, *show=False*)
Set current plot style e.g. "acad.ctb", and *show* impact of plot style also on screen.

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

Access existing entities

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

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

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

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

Parameters

- **dxfattrib** (*str*) – grouping DXF attribute like 'layer'
- **key** (*function*) – key function, which accepts a DXFEntity as argument, returns grouping key of this entity or None for ignore this object. Reason for ignoring: a queried DXF attribute is not supported by this entity

Create new entities

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

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

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

Layout.**add_ellipse**(*center*, *major_axis=(1, 0, 0)*, *ratio=1*, *start_param=0*, *end_param=6.283185307*, *dxfattribs=None*)
Add an `Ellipse` element, *center* is 2D/3D point, *major_axis* as vector, *ratio* is the ratio of minor axis to major axis, *start_param* and *end_param* defines start and end point of the ellipse, a full ellipse goes from 0 to 2*pi. The ellipse goes from start to end param in *counter clockwise* direction.

Layout.**add_arc**(*center*, *radius*, *start_angle*, *end_angle*, *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)

AutoCAD creates a spline through fit points by a proprietary algorithm. *ezdxf* can not reproduce the control point calculation.

Layout.**add_open_spline**(*control_points*, *degree=3*, *dxfattribs=None*)
Add an open uniform *Spline*, *control_points* has to be a list (container or generator) of (x, y, z) tuples, *degree* specifies degree of spline. (requires DXF version AC1015 or later)

Open uniform B-splines start and end at your first and last control points.

Layout.**add_closed_spline**(*control_points*, *degree=3*, *dxfattribs=None*)
Add a closed uniform *Spline*, *control_points* has to be a list (container or generator) of (x, y, z) tuples, *degree* specifies degree of spline. (requires DXF version AC1015 or later)

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

Layout.**add_rational_spline**(*control_points*, *weights*, *degree=3*, *dxfattribs=None*)
Add an open rational uniform *Spline*, *control_points* has to be a list (container or generator) of (x, y, z) tuples, *weights* has to be a list of values, which defines the influence of the associated control point, therefor count of control points has to be equal to the count of weights, *degree* specifies degree of spline. (requires DXF version AC1015 or later)

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

Layout.**add_closed_rational_spline**(*control_points*, *weights*, *degree=3*, *dxfattribs=None*)
Add a closed rational uniform *Spline*, *control_points* has to be a list (container or generator) of (x, y, z) tuples, *weights* has to be a list of values, which defines the influence of the associated control point, therefor count of control points has to be equal to the count of weights, *degree* specifies degree of spline. (requires DXF version AC1015 or later)

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

Layout.**add_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.

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.

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

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

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

GraphicEntity.**copy_to_layout**(*layout*)
Duplicate entity and add new entity to *layout*.

GraphicEntity.**move_to_layout**(*layout*, *source=None*)
Move entity from actual layout to *layout*. For DXF R12 providing *source* is faster, if the entity resides in a block layout, because ezdxf has to search in all block layouts, else *source* is not required.

GraphicEntity.**get_dxf_attrib**(*key*, *default=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.

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

GraphicEntity.**update_attribs**(*dxfattribs*)
Set DXF attributes by a dict() like {'layer': 'test', 'color': 4}.

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

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

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

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_QUADRATIC_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.

DXF Attr	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
clip_mode	R2010	Clip mode: 0 = Outside; 1 = Inside

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

Image.**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

4.3.5 Blocks

Blocks Section

The *BlocksSection* class manages all block definitions of a drawing document.

class BlocksSection

BlocksSection.**__iter__**()
Iterate over all block definitions, yielding *BlockLayout* objects.

BlocksSection.**__contains__**(*entity*)
Test if *BlocksSection* contains the block definition *entity*, *entity* can be a block name as *str* or the *Block* definition itself.

BlocksSection.**__getitem__**(*name*)
Get the *Block* definition by *name*, raises *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 *Attrib* object.

Example for appending an attribute to an INSERT entity with none standard alignment:

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

`Insert.`**`delete_attrib`**(*tag*, *ignore=False*)
Delete an *Attrib* from *Insert*. If *ignore* is *False*, an *KeyError* exception is raised, if *Attrib* *tag* does not exist.

`Insert.`**`delete_all_attribs`**()
Delete all attached *Attrib* entities.

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()*.

4.3.6 Groups

Group

A group is just a bunch of DXF entities tied together. All entities of a group has to be on the same layout (model space or any paper layout but not block). Groups can be named or unnamed, but in reality an unnamed groups has just a special name like '*Annnn'. The name of a group has to be unique in the drawing. Groups are organized in the main group table, which is an *Drawing.groups* of the class *Drawing*.

Group entities have to be in model space or any paper layout but not in a block definition!

class DXFGroup

DXFAttr	Version	Description
description	R13	group description (string)
unnamed	R13	1 for unnamed, 0 for named group (int)
selectable	R13	1 for selectable, 0 for not selectable group (int)

The group name is not stored in the GROUP entity, it is stored in the *DXFGroupTable* object.

DXFGroup.**__iter__**()
Iterate over all DXF entities in this group as instances of *GraphicEntity* or inherited (LINE, CIRCLE, ...).

DXFGroup.**__len__**()
Returns the count of DXF entities in this group.

DXFGroup.**__contains__**(*item*)
Returns *True* if item is in this group else *False*. *item* has to be a handle string or an object of type *GraphicEntity* or inherited.

DXFGroup.**handles**()
Generator over all entity handles in this group.

DXFGroup.**get_name**()
Get name of the group as *string*.

DXFGroup.**edit_data**()
Context manager which yields all the group entities as standard Python list:

```
with group.edit_data() as data:
    # add new entities to a group
    data.append(modelspace.add_line((0, 0), (3, 0)))
    # remove last entity from a group
    data.pop()
```

DXFGroup.**set_data**(*entities*)
Set *entities* as new group content, entities should be iterable and yields instances of *GraphicEntity* or inherited (LINE, CIRCLE, ...).

DXFGroup.**extend**(*entities*)
Append *entities* to group content, entities should be iterable and yields instances of *GraphicEntity* or inherited (LINE, CIRCLE, ...).

DXFGroup.**clear**()
Remove all entities from group.

DXFGroup.**remove_invalid_handles**()
Remove invalid handles from group. Invalid handles: deleted entities, entities in a block layout (but not implemented yet)

GroupTable

There only exists one group table in each drawing, which is accessible by the attribute *Drawing.groups*.

class DXFGroupTable

DXFGroupTable.**__iter__**()
Iterate over all existing groups as *(name, group)* tuples. *name* is the name of the group as *string* and *group* is an object of type *DXFGroup*.

DXFGroupTable.**groups**()
Generator over all existing groups, yields just objects of type *DXFGroup*.

DXFGroupTable.**__len__**()
Returns the count of DXF groups.

DXFGroupTable.**__contains__**(*name*)
Returns *True* if a group *name* exists else *False*.

DXFGroupTable.**get**(*name*)
Returns the group *name* as *DXFGroup* object. Raises *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.

4.3.7 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.

4.3.8 Data Query

Name Query String

A name query string is just a standard regular expression see: http://docs.python.org/3/library/re.html

A `'$'` will be appended to the query string.

For general usage of the query features see the tutorial: *Tutorial for Getting Data from DXF Files*

Entity Query String

```
QueryString := EntityQuery ("[" AttribQuery "]" "i"?)*
```

The query string is the combination of two queries, first the required entity query and second the *optional* attribute query, enclosed in square brackets, append `'i'` after the closing square bracket to ignore case for strings.

Entity Query

The entity query is a whitespace separated list of DXF entity names or the special name `'*'`. Where `'*'` means all DXF entities, all other DXF names have to be uppercase.

Attribute Query

The *optional* attribute query is a boolean expression, supported operators are:

- not (!): !term is true, if term is false
- and (&): term & term is true, if both terms are true
- or (|): term | term is true, if one term is true
- and arbitrary nested round brackets
- append (i) after the closing square bracket to ignore case for strings

Attribute selection is a term: "name comparator value", where name is a DXF entity attribute in lowercase, value is a integer, float or double quoted string, valid comparators are:

- `"=="` equal "value"
- `"!="` not equal "value"
- `"<"` lower than "value"
- `"<="` lower or equal than "value"
- `">"` greater than "value"
- `">="` greater or equal than "value"
- `"?"` match regular expression "value"
- `"!?"` does not match regular expression "value"

Query Result

The *EntityQuery* class is the return type of all `query()` methods. *EntityQuery* contains all DXF entities of the source collection, which matches one name of the entity query AND the whole attribute query. If a DXF entity does not have or support a required attribute, the corresponding attribute search term is false.

examples:

```
'LINE[text ? ".*"]' is always empty, because the LINE entity has no text attribute.

'LINE CIRCLE[layer=="construction"]' => all LINE and CIRCLE entities on layer
↪"construction"
'*[!(layer=="construction" & color<7)]' => all entities except those on layer ==
↪"construction" and color < 7
'*[layer=="construction"]i' => (ignore case) all entities with layer == "construction
↪" | "Construction" | "ConStruction" ...
```

EntityQuery Class

class EntityQuery(*Sequence*)

The *EntityQuery* class is a result container, which is filled with dxf entities matching the query string. It is possible to add entities to the container (extend), remove entities from the container and to filter the container. Supports the standard sequence methods and protocols. (Python Sequence Docs)

EntityQuery.**__init__**(*entities*, *query='*'*)

Setup container with entities matching the initial query.

Parameters

- **entities** – sequence of wrapped DXF entities (at least *GraphicEntity* class)
- **query** (*str*) – *entity query string*

EntityQuery.**extend**(*entities*, *query='*'*, *unique=True*)

Extent the query container by entities matching an additional query.

EntityQuery.**remove**(*query='*'*)

Remove all entities from result container matching this additional query.

EntityQuery.**query**(*query='*'*)

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

EntityQuery.**groupby**(*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*.

4.3.9 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.

4.3.10 Algebra Utilities

This utilities are located at *ezdxf.algebra*, import:

```
from ezdxf.algebra import Vector
```

Functions

ezdxf.algebra.**is_close**(*a*, *b*)
Returns True if value is close to value b, uses `math.isclose(a, b, abs_tol=1e-9)` for Python 3, and emulates this function for Python 2.7.

ezdxf.algebra.**is_close_points**(*p1*, *p2*)
Returns True if all axis of p1 and p2 are close.

ezdxf.algebra.**bspline_control_frame**(*fit_points*, *degree=3*, *method='distance'*, *power=.5*)
Generates the control points for the B-spline control frame by Curve Global Interpolation. Given are the fit points and the degree of the B-spline. The function provides 3 methods for generating the parameter vector t:

1. method = `uniform`, creates a uniform t vector, form 0 to 1 evenly spaced; see uniform method
2. method = `distance`, creates a t vector with values proportional to the fit point distances, see chord length method
3. method = `centripetal`, creates a t vector with values proportional to the fit point distances^power; see centripetal method

Parameters

- **fit_points** – fit points of B-spline, as list of (x, y[, z]) tuples
- **degree** – degree of B-spline
- **method** – calculation method for parameter vector t
- **power** – power for centripetal method

Returns a *BSpline* object, with *BSpline.control_points* containing the calculated control points, also *BSpline.knot_values()* returns the used knot values.

Vector

class ezdxf.algebra.**Vector**

This is an immutable universal 3d vector object. This class is optimized for universality not for speed. Immutable means you can't change (x, y, z) components after initialization:

```
v1 = Vector(1, 2, 3)
v2 = v1
v2.z = 7  # this is not possible, raises AttributeError
v2 = Vector(v2.x, v2.y, 7)  # this creates a new Vector() object
assert v1.z == 3  # and v1 remains unchanged
```

Vector initialization:

- Vector(), returns Vector(0, 0, 0)
- Vector((x, y)), returns Vector(x, y, 0)
- Vector((x, y, z)), returns Vector(x, y, z)

- Vecotr(x, y), returns Vector(x, y, 0)
- Vector(v1, v2), returns v2 - v1 as Vector(x, y, z), where v1, v2 are Vector() objects
- Vector((x1, y1, z1), (x2, y2, z2)), returns Vector(x2, y2, z2) - Vector(x1, y1, z1) as Vector() object
- Vector(x, y, z), returns Vector(x, y, z)

Addition, subtraction, scalar multiplication and scalar division left and right handed are supported:

```
v = Vector(1, 2, 3)
v + (1, 2, 3) == Vector(2, 4, 6)
(1, 2, 3) + v == Vector(2, 4, 6)
v - (1, 2, 3) == Vector(0, 0, 0)
(1, 2, 3) - v == Vector(0, 0, 0)
v * 3 == Vector(3, 6, 9)
3 * v == Vector(3, 6, 9)
Vector(3, 6, 9) / 3 == Vector(1, 2, 3)
-Vector(1, 2, 3) == (-1, -2, -3)
```

Comparision between vectors and vectors to tuples is supported:

```
Vector(1, 2, 3) < Vector (2, 2, 2)
(1, 2, 3) < tuple(Vector(2, 2, 2))  # conversion necessary
Vector(1, 2, 3) == (1, 2, 3)

bool(Vector(1, 2, 3)) is True
bool(Vector(0, 0, 0)) is False
```

Vector Attributes

`Vector.`**`x`**

`Vector.`**`y`**

`Vector.`**`z`**

`Vector.`**`xy`**

Returns Vector (x, y, 0)

`Vector.`**`tup2`**

Returns (x, y) tuple

`Vector.`**`tup3`**

Returns (x, y, z) tuple

`Vector.`**`magnitude`**

Returns length of vector

`Vector.`**`magnitude_square`**

Returns square length of vector

`Vector.`**`is_null`**

Returns True for Vector(0, 0, 0) else False

`Vector.spatial_angle_rad:`

Returns spatial angle between vector and x-axis in radians

Vector.spatial_angle_deg:

Returns spatial angle between vector and x-axis in degrees

Vector.**angle_rad**

Returns angle of vector in the xy-plane in radians.

Vector.**angle_deg**

Returns angle of vector in the xy-plane in degrees.

Vector Methods

Vector.generate(items):

Static method returns generator of Vector() objects created from items.

Vector.list(items):

Static method returns list of Vector() objects created from items.

Vector.**from_rad_angle**(*angle*, *length=1.*)

Static method returns Vector() from angle scaled by length, angle in radians.

Vector.**from_deg_angle**(*angle*, *length=1.*)

Static method returns Vector() from angle scaled by length, angle in degree.

Vector.**__str__**()

Return `(x, y, z)` as string.

Vector.**__repr__**()

Return `Vector(x, y, z)` as string.

Vector.**__len__**()

Returns always 3

Vector.__hash__():

Vector.**copy**()

Returns copy of vector.

Vector.**__copy__**()

Support for copy.copy().

Vector.**__deepcopy__**(*memodict*)

Support for copy.deepcopy().

Vector.**__getitem__**(*index*)

Support for indexing `v[0] == v.x; v[1] == v.y; v[2] == v.z;`

Vector.**__iter__**()

Support for the Python iterator protocol.

Vector.**__abs__**()

Returns length (magnitude) of vector.

Vector.**orthogonal**(*ccw=True*)

Returns orthogonal 2D vector, z value is unchanged.

param ccw counter clockwise if True else clockwise

Vector.**lerp**(*other*, *factor=.5*)

Linear interpolation between vector and other, returns new Vector() object.

param other target vector/point

param factor interpolation factor (0==self, 1=other, 0.5=mid point)

Vector.**project**(*other*)

Project vector other onto self, returns new Vector() object.

Vector.**normalize**(*length=1*)

Returns new normalized Vector() object, optional scaled by length.

Vector.**reversed**(*self*)

Returns -vector as new Vector() object

Vector.**__neg__**()

Returns -vector as new Vector() object

Vector.**__bool__**()

Returns True if vector != (0, 0, 0)

Vector.**__eq__**(*other*)

Vector.**__lt__**(*other*)

Vector.**__add__**(*other*)

Vector.**__radd__**(*other*)

Vector.**__sub__**(*other*)

Vector.**__rsub__**(*other*)

Vector.**__mul__**(*other*)

Vector.**__rmul__**(*other*)

Vector.**__truediv__**(*other*)

Vector.**__div__**(*other*)

Vector.**__rtruediv__**(*other*)

Vector.**__rdiv__**(*other*)

Vector.**dot**(*other*)

Returns 'dot' product of vector . other.

Vector.**cross**(*other*)

Returns 'cross' product of vector x other

Vector.**distance**(*other*)

Returns distance between vector and other.

Vector.**angle_between**(*other*)

Returns angle between vector and other in th xy-plane in radians. +angle is counter clockwise orientation.

Vector.**rot_z_rad**(*angle*)

Return rotated vector around z axis, angle in radians.

Vector.**rot_z_deg**(*angle*)

Return rotated vector around z axis, angle in degrees.

Matrix44

class ezdxf.algebra.**Matrix44**

This is a pure Python implementation for 4x4 transformation matrices, to avoid dependency to big numerical packages like numpy, and before binary wheels, installation of these packages wasn't always easy on Windows.

Matrix44 initialization:

- Matrix44() is the identity matrix.
- Matrix44(values) values is an iterable with the 16 components of the matrix.
- Matrix44(row1, row2, row3, row4) four rows, each row with four values.

Matrix44.**set**(**args*)

Reset matrix values:

- set() creates the identity matrix.
- set(values) values is an iterable with the 16 components of the matrix.
- set(row1, row2, row3, row4) four rows, each row with four values.

Matrix44.**__repr__**()

Returns the representation string of the matrix:

```
Matrix44((col0, col1, col2, col3), (...), (...), (...))
```

Matrix44.**get_row**(*row*)

Get row as list of of four float values.

Matrix44.**set_row**(*row*, *values*)

Sets the values in a row.

param row row index [0..3]

param values four column values as iterable.

Matrix44.**get_col**(*col*)

Get column as list of of four float values.

Matrix44.**set_col**(*col*, *values*)

Sets the values in a column.

param col column index [0..3]

param values four column values as iterable.

Matrix44.**copy**()

Matrix44.**__copy__**()

Matrix44.identity(cls):

Class method to create an identity matrix.

Matrix44.**scale**(*sx*, *sy=None*, *sz=None*)

Class method returns a scaling transformation matrix. If sy is None, sy = sx, and if sz is None sz = sx.

Matrix44.**translate**(*x*, *y*, *z*)

Class method returns a translation matrix to (x, y, z).

Matrix44.**x_rotate**(*angle*)

Class method returns a rotation matrix about the x-axis.

param angle rotation angle in radians

Matrix44.**y_rotate**(*angle*)

Class method returns a rotation matrix about the y-axis.

param angle rotation angle in radians

Matrix44.**z_rotate**(*angle*)

Class method returns a rotation matrix about the z-axis.

param angle rotation angle in radians

Matrix44.**axis_rotate**(*axis*, *angle*)

Class method returns a rotation matrix about an arbitrary axis.

param axis rotation axis as (x, y, z) tuple

angle: rotation angle in radians

Matrix44.**xyz_rotate**(*angle_x*, *angle_y*, *angle_z*)

Class method returns a rotation matrix for rotation about each axis.

param angle_x rotation angle about x-axis in radians

param angle_y rotation angle about y-axis in radians

param angle_z rotation angle about z-axis in radians

Matrix44.**perspective_projection**(*left*, *right*, *top*, *bottom*, *near*, *far*)

Class method returns a matrix for a 2d projection.

param left Coordinate of left of screen

param right Coordinate of right of screen

param top Coordinate of the top of the screen

param bottom Coordinate of the bottom of the screen

param near Coordinate of the near clipping plane

param far Coordinate of the far clipping plane

Matrix44.**perspective_projection_fov**(*fov*, *aspect*, *near*, *far*)

Class method returns a matrix for a 2d projection.

param fov The field of view (in radians)

param aspect The aspect ratio of the screen (width / height)

param near Coordinate of the near clipping plane

param far Coordinate of the far clipping plane

Matrix44.**chain**(**matrices*)

Compose a transformation matrix from one or more matrices.

Matrix44.**__setitem__**(*coord*, *value*)

Set (row, column) element.

Matrix44.**__getitem__**(*coord*)

Get (row, column) element.

Matrix44.**__iter__**()

Iterates over all matrix values.

Matrix44.**__mul__**(*other*)

Returns a new matrix as result of the matrix multiplication with another matrix.

Matrix44.**__imul__**(*other*)

Inplace multiplication with another matrix.

Matrix44.**fast_mul**(*other*)

Multiplies this matrix with other matrix inplace.

Assumes that both matrices have a right column of (0, 0, 0, 1). This is True for matrices composed of rotations, translations and scales. fast_mul is approximately 25% quicker than __imul__().

Matrix44.**rows**()

Iterate over rows as 4-tuples.

Matrix44.**columns**()

Iterate over columns as 4-tuples.

Matrix44.**transform**(*vector*)

Transforms a 3d vector and return the result as a tuple.

Matrix44.**transform_vectors**(*vectors*)

Returns a list of transformed vectors.

Matrix44.**transpose**()

Swaps the rows for columns inplace.

Matrix44.**get_transpose**()

Returns a new transposed matrix.

Matrix44.**determinant**()

Returns determinant.

Matrix44.**inverse**()

Returns the inverse of the matrix.

Raises ZeroDivisionError if matrix has no inverse.

BSpline

class ezdxf.algebra.**BSpline**
Calculate the vertices of a B-spline curve, using an uniform open knot vector (clamped curve).

BSpline.**control_points**
Control points as list of *Vector* objects

BSpline.**count**
Count of control points, (n + 1 in math definition).

BSpline.**order**
Order of B-spline = degree + 1

BSpline.**degree**
Degree (p) of B-spline = order - 1

BSpline.**max_t**
Max knot value.

BSpline.**knot_values**()
Returns a list of knot values as floats, the knot vector always has order+count values (n + p + 2 in math definition)

BSpline.**basis_values**(*t*)
Returns the basis vector for position t.

BSpline.**approximate**(*segments*)
Approximates the whole B-spline from 0 to max_t, by line segments as a list of vertices, vertices count = segments + 1

BSpline.**point**(*t*)
Returns the B-spline vertex at position t as (x, y[, z]) tuple.

BSplineU

class ezdxf.algebra.**BSpline**(*BSpline*)
Calculate the points of a B-spline curve, uniform (periodic) knot vector (open curve).

BSplineClosed

class ezdxf.algebra.**BSplineClosed**(*BSplineU*)
Calculate the points of a closed uniform B-spline curve (closed curve).

DBSpline

class ezdxf.algebra.**DBSpline**(*BSpline*)
Calculate points and derivative of a B-spline curve, using an uniform open knot vector (clamped curve).

DBSpline.**point**(*t*)
Returns the B-spline vertex, 1. derivative and 2. derivative at position t as tuple (vertex, d1, d2), each value is a (x, y, z) tuple.

DBSplineU

class ezdxf.algebra.**DBSplineU**(*DBSpline*)
Calculate points and derivative of a B-spline curve, uniform (periodic) knot vector (open curve).

DBSplineClosed

class ezdxf.algebra.**DBSplineClosed**(*DBSplineU*)
Calculate the points and derivative of a closed uniform B-spline curve (closed curve).

4.4 Add-ons

TODO

4.5 Howto

General preconditions:

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

4.5.1 Get/Set block reference attributes

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

Iterate over all appended attributes:

```
blockrefs = modelspace.query('INSERT[name=="Part12"]')  # get all INSERT entities␣
↪with entity.dxf.name == "Part12"
if len(blockrefs):
    entity = blockrefs[0]  # process first entity found
    for attrib in entity.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"
```

4.5.2 Reduce Memory Footprint

- compress binary data by *Drawing.compress_binary_data()*

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

4.5.3 Create More Readable DXF Files (DXF Pretty Printer)

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

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

```
# on Windows
py -3 -m ezdxf.pp your_dxf_file.dxf

# on Linux/Mac
python3 -m ezdxf.pp your_dxf_file.dxf
```

This produces a HTML file *your_dxf_file.html* with a nicer layout than a plain DXF file and DXF handles as links between DXF entities, this simplifies the navigation between the DXF entities.

Since ezdxf v0.8.3, a script called *dxfpp* will be added to your Python script path:

```
usage: dxfpp [-h] [-o] [-r] [-x] [-l] FILE [FILE ...]

positional arguments:
  FILE             DXF files pretty print

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

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

4.5.4 Adding New XDATA to Entity

Adding XDATA as list of tuples (group code, value):

```
dwg.appids.new('YOUR_APP_NAME')  # IMPORTANT: create an APP ID entry

circle = modelspace.add_circle((10, 10), 100)
circle.tags.new_xdata('YOUR_APP_NAME',
                  [
                      (1000, 'your_web_link.org'),
                      (1002, '{'),
                      (1000, 'some text'),
                      (1002, '{'),
                      (1071, 1),
                      (1002, '}'),
                      (1002, '}')
                  ])
```

For group code meaning see DXF reference section *DXF Group Codes in Numerical Order Reference*, valid group codes are in the range 1000 - 1071.

4.5.5 A360 Viewer Problems

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

4.5.6 Show IMAGES/XREFS on Loading in AutoCAD

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

Thanks to David Booth:

> If the filename in the IMAGEDEF contains the full path (absolute in AutoCAD) then it shows on loading, otherwise it won't display (reports as unreadable) until you manually reload using XREF manager.
>
> A workaround (to show IMAGES on loading) appears to be to save the full file path in the DXF or save it as a DWG.

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

4.6 DXF Internals

- DXF Reference provided by Autodesk.
- DXF Developer Documentation provided by Autodesk.

4.6.1 DXF File Encoding

DXF Version R2004 and prior

Drawing files of DXF versions R2004 (AC1018) and prior are saved as ASCII files with the encoding set by the header variable *$DWGCODEPAGE*, which is ANSI_1252 by default if *$DWGCODEPAGE* is not set.

Characters used in the drawing which do not exist in the chosen ASCII encoding are encoded as unicode characters with the schema `\U+nnnn`. see Unicode table

Known *$DWGCODEPAGE* encodings

DXF	Python	Name
ANSI_874	cp874	Thai
ANSI_932	cp932	Japanese
ANSI_936	gbk	UnifiedChinese
ANSI_949	cp949	Korean
ANSI_950	cp950	TradChinese
ANSI_1250	cp1250	CentralEurope
ANSI_1251	cp1251	Cyrillic
ANSI_1252	cp1252	WesternEurope
ANSI_1253	cp1253	Greek
ANSI_1254	cp1254	Turkish
ANSI_1255	cp1255	Hebrew
ANSI_1256	cp1256	Arabic
ANSI_1257	cp1257	Baltic
ANSI_1258	cp1258	Vietnam

DXF Version R2007 and later

Starting with DXF version R2007 (AC1021) the drawing file is encoded by UTF-8, the header variable *$DWGCODEPAGE* is still in use, but I don't know, if the setting still has any meaning.

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

See also:

String Value Encoding

4.6.2 DXF Tags

A Drawing Interchange File is simply an ASCII text file with a file type of .dxf and special formatted text. The basic file structure are DXF tags, a DXF tag consist of a DXF group code as an integer value on its own line and a the DXF value on the following line. In the ezdxf documentation DXF tags will be written as `(group code, value)`.

Group codes are indicating the value type:

Group Code	Value Type
0-9	String (with the introduction of extended symbol names in DXF R2000, the 255-character limit has been increased to
10-39	Double precision 3D point value
40-59	Double-precision floating-point value
40-59	Double-precision floating-point value
60-79	16-bit integer value
90-99	32-bit integer value
100	String (255-character maximum, less for Unicode strings)
102	String (255-character maximum, less for Unicode strings)
105	String representing hexadecimal (hex) handle value
110-119	Double precision floating-point value
120-129	Double precision floating-point value
130-139	Double precision floating-point value
140-149	Double precision scalar floating-point value
160-169	64-bit integer value
170-179	16-bit integer value
210-239	Double-precision floating-point value
270-279	16-bit integer value
280-289	16-bit integer value
290-299	Boolean flag value
300-309	Arbitrary text string
310-319	String representing hex value of binary chunk
320-329	String representing hex handle value
330-369	String representing hex object IDs
370-379	16-bit integer value
380-389	16-bit integer value
390-399	String representing hex handle value
400-409	16-bit integer value
410-419	String
420-429	32-bit integer value
430-439	String
440-449	32-bit integer value
450-459	Long
460-469	Double-precision floating-point value

Table 4.1 – continued from previous page

Group Code	Value Type
470-479	String
480-481	String representing hex handle value
999	Comment (string)
1000-1009	String (same limits as indicated with 0-9 code range)
1010-1059	Double-precision floating-point value
1060-1070	16-bit integer value
1071	32-bit integer value

Explanation for some important group codes:

Group Code	Meaning
0	DXF structure tag, entity start/end or table entries
1	The primary text value for an entity
2	A name: Attribute tag, Block name, and so on. Also used to identify a DXF section or table name.
3-4	Other textual or name values
5	Entity handle expressed as a hex string (fixed)
6	Line type name (fixed)
7	Text style name (fixed)
8	Layer name (fixed)
9	Variable name identifier (used only in HEADER section of the DXF file)
10	Primary X coordinate (start point of a Line or Text entity, center of a Circle, etc.)
11-18	Other X coordinates
20	Primary Y coordinate. 2n values always correspond to 1n values and immediately follow them in the file (expected by ezdxf!)
21-28	Other Y coordinates
30	Primary Z coordinate. 3n values always correspond to 1n and 2n values and immediately follow them in the file (expected by ezdxf!)
31-38	Other Z coordinates
39	This entity's thickness if nonzero (fixed)
40-48	Float values (text height, scale factors, etc.)
49	Repeated value - multiple 49 groups may appear in one entity for variable length tables (such as the dash lengths in the LTYPE table). A 7x group always appears before the first 49 group to specify the table length
50-58	Angles
62	Color number (fixed)
66	"Entities follow" flag (fixed), only in INSERT and POLYLINE entities
67	Identifies whether entity is in model space or paper space
68	Identifies whether viewport is on but fully off screen, is not active, or is off
69	Viewport identification number
70-78	Integer values such as repeat counts, flag bits, or modes
210, 220, 230	X, Y, and Z components of extrusion direction (fixed)
999	Comments

For explanation of all group codes see: DXF Group Codes in Numerical Order Reference provided by Autodesk

Extended Data

Extended data (xdata) is created by AutoLISP or ObjectARX applications but any other application like ezdxf can also define xdata. If an entity contains extended data, it **follows** the entity's normal definition data.

Group Code	Description
1000	Strings in extended data can be up to 255 bytes long (with the 256th byte reserved for the null character)
1001	(fixed) Registered application name (ASCII string up to 31 bytes long) for XDATA
1002	(fixed) An extended data control string can be either "{"or "}". These braces enable applications to organize their data by subdividing the data into lists. Lists can be nested.
1003	Name of the layer associated with the extended data
1004	Binary data is organized into variable-length chunks. The maximum length of each chunk is 127 bytes. In ASCII DXF files, binary data is represented as a string of hexadecimal digits, two per binary byte
1005	Database Handle of entities in the drawing database, see also: *About 1005 Group Codes*
1010, 1020, 1030	Three real values, in the order X, Y, Z. They can be used as a point or vector record.
1011, 1021, 1031	Unlike a simple 3D point, the world space coordinates are moved, scaled, rotated, mirrored, and stretched along with the parent entity to which the extended data belongs.
1012, 1012, 1022	Also a 3D point that is scaled, rotated, and mirrored along with the parent (but is not moved or stretched)
1013, 1023, 1033	Also a 3D point that is scaled, rotated, and mirrored along with the parent (but is not moved or stretched)
1040	A real value
1041	Distance, a real value that is scaled along with the parent entity
1042	Scale Factor, also a real value that is scaled along with the parent. The difference between a distance and a scale factor is application-defined
1070	A 16-bit integer (signed or unsigned)
1071	A 32-bit signed (long) integer

The `(1001, ...)` tag indicates the beginning of extended data. In contrast to normal entity data, with extended data the same group code can appear multiple times, and **order is important**.

Extended data is grouped by registered application name. Each registered application group begins with a `(1001, APPID)` tag, with the application name as APPID string value. Registered application names correspond to APPID symbol table entries.

An application can use as many APPID names as needed. APPID names are permanent, although they can be purged if they aren't currently used in the drawing. Each APPID name can have **no more than one data group** attached to each entity. Within an application group, the sequence of extended data groups and their meaning is defined by the application.

String Value Encoding

String values stored in a DXF file is plain ASCII or UTF-8, AutoCAD also supports CIF (Common Interchange Format) and MIF (Maker Interchange Format) encoding. The UTF-8 format is only supported in DXF R2007 and later.

ezdxf on import converts all strings into Python unicode strings without encoding or decoding CIF/MIF.

String values containing Unicode characters are represented with control character sequences.

For example, 'TESTU+7F3AU+4E4FU+89E3U+91CAU+6B63THISU+56FE'

To support the DXF unicode encoding ezdxf registers an encoding codec *dxfbackslashreplace*, defined in `ezdxf.lldxf.encoding()`.

String values can be stored with these dxf group codes:

- 0 - 9
- 100 - 101
- 300 - 309
- 410 - 419
- 430 - 439
- 470 - 479
- 999 - 1003

Multi Tag Text (MTEXT)

If the text string is less than 250 characters, all characters appear in tag `(1, ...)`. If the text string is greater than 250 characters, the string is divided into 250-character chunks, which appear in one or more `(3, ...)` tags. If `(3, ...)` tags are used, the last group is a `(1, ...)` tag and has fewer than 250 characters:

```
3
... TwoHundredAndFifty Characters ....
3
... TwoHundredAndFifty Characters ....
1
less than TwoHundredAndFifty Characters
```

As far I know this is only supported by the MTEXT entity.

See also:

DXF File Encoding

Tag Structure DXF R13 and later

With the introduction of DXF R13 Autodesk added additional group codes and DXF tag structures to the DXF Standard.

Subclass Markers

Subclass markers `(100, Subclass Name)` divides DXF objects into several sections. Group codes can be reused in different sections. A subclass ends with the following subclass marker or at the beginning of xdata or the end of the object. See Subclass Marker Example in the DXF Reference.

Extension Dictionary

The extension dictionary is an optional sequence that stores the handle of a dictionary object that belongs to the current object, which in turn may contain entries. This facility allows attachment of arbitrary database objects to any database object. Any object or entity may have this section.

The extension dictionary tag sequence:

```
102
{ACAD_XDICTIONARY
360
Hard-owner ID/handle to owner dictionary
102
}
```

Persistent Reactors

Persistent reactors are an optional sequence that stores object handles of objects registering themselves as reactors on the current object. Any object or entity may have this section.

The persistent reactors tag sequence:

```
102
{ACAD_REACTORS
330
first Soft-pointer ID/handle to owner dictionary
330
second Soft-pointer ID/handle to owner dictionary
...
102
}
```

Application-Defined Codes

Starting at DXF R13, DXF objects can contain application-defined codes outside of xdata. This application-defined codes can contain any tag except `(0, ...)` and `(102, ...)`. "{YOURAPPID" means the APPID string with an preceding "{". The application defined data tag sequence:

```
102
{YOURAPPID
...
102
}
```

All groups defined with a beginning `(102, ...)` appear in the DXF reference before the first subclass marker, I don't know if these groups can appear after the first or any subclass marker. ezdxf accepts them at any position, and by default ezdxf adds new app data in front of the first subclass marker to the first tag section of an DXF object.

4.6.3 Handles

A handle is an arbitrary but in your DXF file unique hex value as string like '10FF'. It is common to to use uppercase letters for hex numbers. Handle can have up to 16 hexadecimal digits.

For DXF R10 until R12 the usage of handles was optional. The header variable $HANDLING set to 1 indicate the usage of handles, else $HANDLING is 0 or missing.

For DXF R13 and later the usage of handles is mandatory and the header variable $HANDLING was removed.

The $HANDSEED variable in the header section should be greater than the biggest handle used in the DXF file, so a CAD application can assign handle values starting with the $HANDSEED value. But as always, don't rely on the header variable it could be wrong, AutoCAD ignores this value.

Handle Definition

Entity handle definition is always the `(5, ...)`, except for entities of the DIMSTYLE table `(105, ...)`, because the DIMSTYLE entity has also a group code 5 tag for DIMBLK.

Handle Pointer

A pointer is a reference to a DXF object in the same DXF file. There are four types of pointers:

- Soft-pointer handle
- Hard-pointer handle
- Soft-owner handle
- Hard-owner handle

Also, a group code range for "arbitrary" handles is defined to allow convenient storage of handle values that are unchanged at any operation (AutoCAD).

Pointer and Ownership

A pointer is a reference that indicates usage, but not possession or responsibility, for another object. A pointer reference means that the object uses the other object in some way, and shares access to it. An ownership reference means that an owner object is responsible for the objects for which it has an owner handle. An object can have any number of pointer references associated with it, but it can have only one owner.

Hard and Soft References

Hard references, whether they are pointer or owner, protect an object from being purged. Soft references do not.

In AutoCAD, block definitions and complex entities are hard owners of their elements. A symbol table and dictionaries are soft owners of their elements. Polyline entities are hard owners of their vertex and seqend entities. Insert entities are hard owners of their attrib and seqend entities.

When establishing a reference to another object, it is recommended that you think about whether the reference should protect an object from the PURGE command.

Arbitrary Handles

Arbitrary handles are distinct in that they are not translated to session-persistent identifiers internally, or to entity names in AutoLISP, and so on. They are stored as handles. When handle values are translated in drawing-merge operations, arbitrary handles are ignored.

In all environments, arbitrary handles can be exchanged for entity names of the current drawing by means of the handent functions. A common usage of arbitrary handles is to refer to objects in external DXF and DWG files.

About 1005 Group Codes

`(1005, ...)` xdata have the same behavior and semantics as soft pointers, which means that they are translated whenever the host object is merged into a different drawing. However, 1005 items are not translated to session-persistent identifiers or internal entity names in AutoLISP and ObjectARX. They are stored as handles.

4.6.4 DXF File Structure

A DXF File is simply an ASCII text file with a file type of .dxf and special formatted text. The basic file structure are DXF tags, a DXF tag consist of a DXF group code as an integer value on its own line and a the DXF value on the following line. In the ezdxf documentation DXF tags will be written as `(group code, value)`. I know there exists a binary DXF format, but it seems that it is not often used and for reducing file size, zipping is much more efficient. ezdxf does not support binary encoded DXF files (yet?).

See also:

For more information about DXF tags see: *DXF Tags*

A usual DXF file is organized in sections, starting with the DXF tag (0, 'SECTION') and ending with the DXF tag (0, 'ENDSEC'). The (0, 'EOF') tag signals the end of file.

1. **HEADER:** General information about the drawing is found in this section of the DXF file. Each parameter has a variable name starting with '$' and an associated value. Has to be the first section.
2. **CLASSES:** Holds the information for application defined classes. (DXF R13 and later)
3. **TABLES:**: Contains several tables for style and property definitions.
   - Linetype table (LTYPE)
   - Layer table (LAYER)
   - Text Style table (STYLE)
   - View table (VIEW): (IMHO) layout of the CAD working space, only interesting for interactive CAD applications
   - Viewport configuration table (VPORT): The VPORT table is unique in that it may contain several entries with the same name (indicating a multiple-viewport configuration). The entries corresponding to the active viewport configuration all have the name *ACTIVE. The first such entry describes the current viewport.
   - Dimension Style table (DIMSTYLE)
   - User Coordinate System table (UCS) (IMHO) only interesting for interactive CAD applications
   - Application Identification table (APPID): Table of names for all applications registered with a drawing.
   - Block Record table (BLOCK_RECORD) (DXF R13 and Later)
4. **BLOCKS:** Contains all block definitions. The block name *Model_Space or *MODEL_SPACE is reserved for the drawing model space and the block name *Paper_Space or *PAPER_SPACE is reserved for the *active* paper space layout. Both block definitions are empty, the content of the model space and the *active* paper space is stored in the ENTITIES section. The entities of other layouts are stored in special block definitions called *Paper_Spacennn, nnn is an arbitrary but unique number.
5. **ENTITIES:** Contains all graphical entities of the model space and the *active* paper space layout. Entities of other layouts are stored in the BLOCKS sections.
6. **OBJECTS:** Contains all non-graphical objects of the drawing (DXF R13 and later)
7. **THUMBNAILIMAGE:** Contains a preview image of the DXF file, it is optional and can usually be ignored. (DXF R13 and later)

8. **ACDSDATA:** (DXF R2013 and later) No information in the DXF reference about this section
9. **END OF FILE**

For further information read the original DXF Reference.

Structure of a usual DXF R12 file:

```
0           <<< Begin HEADER section, has to be the first section
SECTION
2
HEADER
            <<< Header variable items go here
0           <<< End HEADER section
ENDSEC
0           <<< Begin TABLES section
SECTION
2
TABLES
0
TABLE
2
VPORT
70          <<< viewport table maximum item count
            <<< viewport table items go here
0
ENDTAB
0
TABLE
2
APPID, DIMSTYLE, LTYPE, LAYER, STYLE, UCS, VIEW, or VPORT
70          <<< Table maximum item count, a not reliable value and ignored by AutoCAD
            <<< Table items go here
0
ENDTAB
0           <<< End TABLES section
ENDSEC
0           <<< Begin BLOCKS section
SECTION
2
BLOCKS
            <<< Block definition entities go here
0           <<< End BLOCKS section
ENDSEC
0           <<< Begin ENTITIES section
SECTION
2
ENTITIES
            <<< Drawing entities go here
0           <<< End ENTITIES section
ENDSEC
0           <<< End of file marker (required)
EOF
```

4.6.5 Minimal DXF Content

DXF R12

Contrary to the previous chapter, the DXF R12 format (AC1009) and prior requires just the ENTITIES section:

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

DXF R13/R14 and later

DXF version R13/14 and later needs much more DXF content than DXF R12.

Required sections: HEADER, CLASSES, TABLES, ENTITIES, OBJECTS

The HEADER section requires two entries:

- $ACADVER
- $HANDSEED

The CLASSES section can be empty, but some DXF entities requires class definitions to work in AutoCAD.

The TABLES section requires following tables:

- VPORT entry *ACTIVE is not required! Empty table is ok for AutoCAD.
- LTYPE with at least the following line types defined:
  - BYBLOCK
  - BYLAYER
  - CONTINUOUS
- LAYER with at least an entry for layer '0'
- STYLE with at least an entry for style STANDARD
- VIEW can be empty
- UCS can be empty
- APPID with at least an entry for ACAD
- DIMSTYLE with at least an entry for style STANDARD
- BLOCK_RECORDS with two entries:
  - *MODEL_SPACE
  - *PAPER_SPACE

The BLOCKS section requires two BLOCKS:

- *MODEL_SPACE
- *PAPER_SPACE

The ENTITIES section can be empty.

The OBJECTS section requires following entities:

- DICTIONARY - the root dict - one entry named ACAD_GROUP
- DICTONARY ACAD_GROUP can be empty

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

4.6.6 Coordinate Systems

AutoLISP Reference to Coordinate Systems provided by Autodesk.

WCS

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

UCS

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

OCS

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

DCS

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

4.6.7 HEADER Section

Documentation to ezdxf *HeaderSection* class.

In DXF R12 an prior the HEADER section was optional, but since DXF R13 the HEADER section is mandatory. The overall structure is:

```
0           <<< Begin HEADER section
SECTION
2
HEADER
9
$ACADVER    <<< Header variable items go here
```

```
1
AC1009
...
0
ENDSEC      <<< End HEADER section
```

A header variable has a name defined by a `(9, Name)` tag and following value tags.

See also:

DXF Reference: Header Variables

4.6.8 CLASSES Section

The CLASSES section contains CLASS definitions which are only important for Autodesk products, some DXF entities require a class definition or AutoCAD will not open the DXF file.

The CLASSES sections was introduced with DXF AC1015 (AutoCAD Release R13).

See also:

DXF Reference: About the DXF CLASSES Section

The CLASSES section in DXF files holds the information for application-defined classes whose instances appear in the BLOCKS, ENTITIES, and OBJECTS sections of the database. It is assumed that a class definition is permanently fixed in the class hierarchy. All fields are required.

CLASS Entities

See also:

DXF Reference: Group Codes for the CLASS entity

CLASS entities have no handle and therefor ezdxf does not store the CLASS entity in the drawing entities database!

```
0
SECTION
2           <<< begin CLASSES section
CLASSES
0           <<< first CLASS entity
CLASS
1           <<< class DXF entity name; always unique
ACDBDICTIONARYWDFLT
2           <<< C++ class name; always unique
AcDbDictionaryWithDefault
3           <<< application name
ObjectDBX Classes
90          <<< proxy capabilities flags
0
91          <<< instance counter for custom class, since DXF version AC1018 (R2004)
0           <<< no problem if the counter is wrong, AutoCAD doesn't care about
280         <<< was-a-proxy flag. Set to 1 if class was not loaded when this DXF file␣
↪was created, and 0 otherwise
0
281         <<< is-an-entity flag. Set to 1 if class reside in the BLOCKS or ENTITIES␣
↪section. If 0, instances may appear only in the OBJECTS section
0
0           <<< second CLASS entity
```

```
CLASS
...
...
0           <<< end of CLASSES section
ENDSEC
```

4.6.9 TABLES Section

VIEW Table

The VIEW entry stores a named view of the model or a paper space layout. This stored views makes parts of the drawing or some view points of the model in a CAD applications more accessible. This views have no influence to the drawing content or to the generated output by exporting PDFs or plotting on paper sheets, they are just for the convenience of CAD application users.

Using *ezdxf* you have access to the views table by the attribute `Drawing.views`. The views table itself is not stored in the entity database, but the table entries are stored in entity database, and can be accessed by its handle.

DXF R12

```
0
VIEW
2       <<< name of view
VIEWNAME
70      <<< flags bit-coded: 1st bit -> (0/1 = model space/paper space)
0       <<< model space
40      <<< view width in Display Coordinate System (DCS)
20.01
10      <<< view center point in DCS
40.36   <<<     x value
20      <<<     group code for y value
15.86   <<<     y value
41      <<< view height in DCS
17.91
11      <<< view direction from target point, 3D vector
0.0     <<<     x value
21      <<<     group code for y value
0.0     <<<     y value
31      <<<     group code for z value
1.0     <<<     z value
12      <<< target point in WCS
0.0     <<<     x value
22      <<<     group code for y value
0.0     <<<     y value
32      <<<     group code for z value
0.0     <<<     z value
42      <<< lens (focal) length
50.0    <<< 50mm
43      <<< front clipping plane, offset from target
0.0
44      <<< back clipping plane, offset from target
0.0
50      <<< twist angle
0.0
```

```
71      <<< view mode
0
```

See also:

Coordinate Systems

DXF R2000+

Mostly the same structure as DXF R12, but with handle, owner tag and subclass markers.

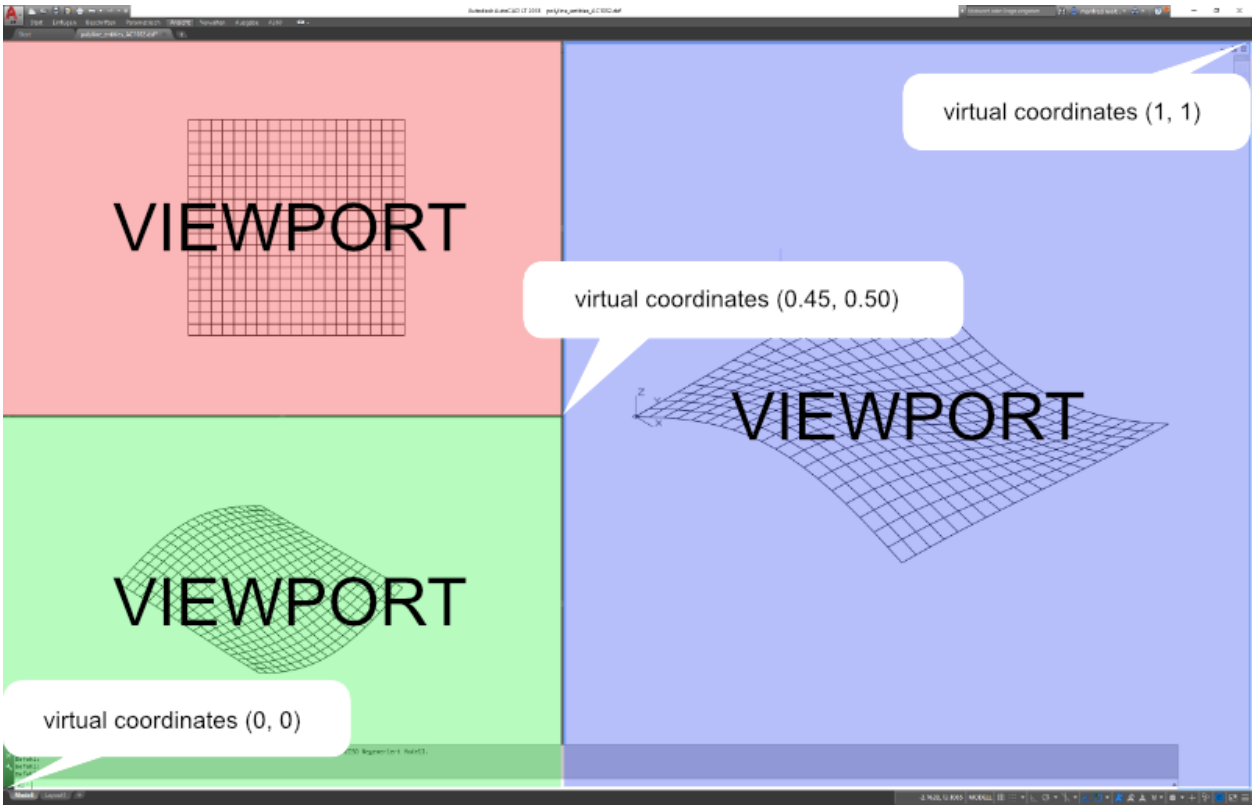
```
0       <<< adding the VIEW table head, just for information
TABLE
2       <<< table name
VIEW
5       <<< handle of table, see owner tag of VIEW table entry
37C
330     <<< owner tag of table, always #0
0
100     <<< subclass marker
AcDbSymbolTable
70      <<< VIEW table (max.) count, not reliable (ignore)
9
0       <<< first VIEW table entry
VIEW
5       <<< handle
3EA
330     <<< owner, the VIEW table is the owner of the VIEW entry
37C     <<< handle of the VIEW table
100     <<< subclass marker
AcDbSymbolTableRecord
100     <<< subclass marker
AcDbViewTableRecord
2       <<< view name, from here all the same as DXF R12
VIEWNAME
70
0
40
20.01
10
40.36
20
15.86
41
17.91
11
0.0
21
0.0
31
1.0
12
0.0
22
0.0
32
0.0
```

```
42
50.0
43
0.0
44
0.0
50
0.0
71
0
281     <<< render mode 0-6 (... too much options)
0       <<< 0= 2D optimized (classic 2D)
72      <<< UCS associated (0/1 = no/yes)
0       <<< 0 = no
```

DXF R2000+ supports additional features in the VIEW entry, see the VIEW table reference provided by Autodesk.

VPORT Configuration Table

The VPORT table stores the model space viewport configurations. A viewport configuration is a tiled view of multiple viewports or just one viewport.



In contrast to other tables the VPORT table can have multiple entries with the same name, because all VPORT entries of a multi-viewport configuration are having the same name - the viewport configuration name. The name of the actual displayed viewport configuration is `*ACTIVE`, as always table entry names are case insensitive (`*ACTIVE` == `*Active`).

The available display area in AutoCAD has normalized coordinates, the lower-left corner is (0, 0) and the upper-right corner is (1, 1) regardless of the true aspect ratio and available display area in pixels. A single viewport configuration

has one VPORT entry `*ACTIVE` with the lower-left corner (0, 0) and the upper-right corner (1, 1).

The following statements refer to a 2D plan view: the view-target-point defines the origin of the DCS (Display Coordinate system), the view-direction vector defines the z-axis of the DSC, the view-center-point (in DCS) defines the point in model space translated to the center point of the viewport, the view height and the aspect-ratio defines how much of the model space is displayed. AutoCAD tries to fit the model space area into the available viewport space e.g. view height is 15 units and aspect-ratio is 2.0 the model space to display is 30 units wide and 15 units high, if the viewport has an aspect ratio of 1.0, AutoCAD displays 30x30 units of the model space in the viewport. If the model space aspect-ratio is 1.0 the model space to display is 15x15 units and fits properly into the viewport area.

But tests show that the translation of the view-center-point to the middle of the viewport not always work as I expected. (still digging...)

Note: All floating point values are rounded to 2 decimal places for better readability.

DXF R12

Multi-viewport configuration with three viewports.

```
0       <<< table start
TABLE
2       <<< table type
VPORT
70      <<< VPORT table (max.) count, not reliable (ignore)
3
0       <<< first VPORT entry
VPORT
2       <<< VPORT (configuration) name
*ACTIVE
70      <<< standard flags, bit-coded
0
10      <<< lower-left corner of viewport
0.45    <<<     x value, virtual coordinates in range [0 - 1]
20      <<<     group code for y value
0.0     <<<     y value, virtual coordinates in range [0 - 1]
11      <<< upper-right corner of viewport
1.0     <<<     x value, virtual coordinates in range [0 - 1]
21      <<<     group code for y value
1.0     <<<     y value, virtual coordinates in range [0 - 1]
12      <<< view center point (in DCS), ???
13.71   <<<     x value
22      <<<     group code for y value
0.02    <<<     y value
13      <<< snap base point (in DCS)
0.0     <<<     x value
23      <<<     group code for y value
0.0     <<<     y value
14      <<< snap spacing X and Y
1.0     <<<     x value
24      <<<     group code for y value
1.0     <<<     y value
15      <<< grid spacing X and Y
0.0     <<<     x value
25      <<<     group code for y value
0.0     <<<     y value
```

```
16      <<< view direction from target point (in WCS), defines the z-axis of the DCS
1.0     <<<     x value
26      <<<     group code for y value
-1.0    <<<     y value
36      <<<     group code for z value
1.0     <<<     z value
17      <<< view target point (in WCS), defines the origin of the DCS
0.0     <<<     x value
27      <<<     group code for y value
0.0     <<<     y value
37      <<<     group code for z value
0.0     <<<     z value
40      <<< view height
35.22
41      <<< viewport aspect ratio
0.99
42      <<< lens (focal) length
50.0    <<< 50mm
43      <<< front clipping planes, offsets from target point
0.0
44      <<< back clipping planes, offsets from target point
0.0
50      <<< snap rotation angle
0.0
51      <<< view twist angle
0.0
71      <<< view mode
0
72      <<< circle zoom percent
1000
73      <<< fast zoom setting
1
74      <<< UCSICON setting
3
75      <<< snap on/off
0
76      <<< grid on/off
0
77      <<< snap style
0
78      <<< snap isopair
0
0       <<< next VPORT entry
VPORT
2       <<< VPORT (configuration) name
*ACTIVE <<< same as first VPORT entry
70
0
10
0.0
20
0.5
11
0.45
21
1.0
12
8.21
```

```
22
9.41
...
...
0       <<< next VPORT entry
VPORT
2       <<< VPORT (configuration) name
*ACTIVE <<< same as first VPORT entry
70
0
10
0.0
20
0.0
11
0.45
21
0.5
12
2.01
22
-9.33
...
...
0
ENDTAB
```

DXF R2000+

Mostly the same structure as DXF R12, but with handle, owner tag and subclass markers.

```
0       <<< table start
TABLE
2       <<< table type
VPORT
5       <<< table handle
151F
330     <<< owner, table has no owner - always #0
0
100     <<< subclass marker
AcDbSymbolTable
70      <<< VPORT table (max.) count, not reliable (ignore)
3
0       <<< first VPORT entry
VPORT
5       <<< entry handle
158B
330     <<< owner, VPORT table is owner of VPORT entry
151F
100     <<< subclass marker
AcDbSymbolTableRecord
100     <<< subclass marker
AcDbViewportTableRecord
2       <<< VPORT (configuration) name
*ACTIVE
70      <<< standard flags, bit-coded
```

```
0
10      <<< lower-left corner of viewport
0.45    <<<     x value, virtual coordinates in range [0 - 1]
20      <<<     group code for y value
0.0     <<<     y value, virtual coordinates in range [0 - 1]
11      <<< upper-right corner of viewport
1.0     <<<     x value, virtual coordinates in range [0 - 1]
21      <<<     group code for y value
1.0     <<<     y value, virtual coordinates in range [0 - 1]
12      <<< view center point (in DCS)
13.71   <<<     x value
22      <<<     group code for y value
0.38    <<<     y value
13      <<< snap base point (in DCS)
0.0     <<<     x value
23      <<<     group code for y value
0.0     <<<     y value
14      <<< snap spacing X and Y
1.0     <<<     x value
24      <<<     group code for y value
1.0     <<<     y value
15      <<< grid spacing X and Y
0.0     <<<     x value
25      <<<     group code for y value
0.0     <<<     y value
16      <<< view direction from target point (in WCS)
1.0     <<<     x value
26      <<<     group code for y value
-1.0    <<<     y value
36      <<<     group code for z value
1.0     <<<     z value
17      <<< view target point (in WCS)
0.0     <<<     x value
27      <<<     group code for y value
0.0     <<<     y value
37      <<<     group code for z value
0.0     <<<     z value
40      <<< view height
35.22
41      <<< viewport aspect ratio
0.99
42      <<< lens (focal) length
50.0    <<< 50mm
43      <<< front clipping planes, offsets from target point
0.0
44      <<< back clipping planes, offsets from target point
0.0
50      <<< snap rotation angle
0.0
51      <<< view twist angle
0.0
71      <<< view mode
0
72      <<< circle zoom percent
1000
73      <<< fast zoom setting
1
74      <<< UCSICON setting
```

```
3
75      <<< snap on/off
0
76      <<< grid on/off
0
77      <<< snap style
0
78      <<< snap isopair
0
281     <<< render mode 1-6 (... too many options)
0       <<< 0 = 2D optimized (classic 2D)
65      <<< Value of UCSVP for this viewport. (0 = UCS will not change when this␣
↪viewport is activated)
1       <<< 1 = then viewport stores its own UCS which will become the current UCS␣
↪whenever the viewport is activated.
110     <<< UCS origin (3D point)
0.0     <<<     x value
120     <<<     group code for y value
0.0     <<<     y value
130     <<<     group code for z value
0.0     <<<     z value
111     <<< UCS X-axis (3D vector)
1.0     <<<     x value
121     <<<     group code for y value
0.0     <<<     y value
131     <<<     group code for z value
0.0     <<<     z value
112     <<< UCS Y-axis (3D vector)
0.0     <<<     x value
122     <<<     group code for y value
1.0     <<<     y value
132     <<<     group code for z value
0.0     <<<     z value
79      <<< Orthographic type of UCS 0-6 (... too many options)
0       <<< 0 = UCS is not orthographic
146     <<< elevation
0.0
1001    <<< extended data - undocumented
ACAD_NAV_VCDISPLAY
1070
3
0       <<< next VPORT entry
VPORT
5
158C
330
151F
100
AcDbSymbolTableRecord
100
AcDbViewportTableRecord
2       <<< VPORT (configuration) name
*ACTIVE <<< same as first VPORT entry
70
0
10
0.0
20
```

```
0.5
11
0.45
21
1.0
12
8.21
22
9.72
...
...
0       <<< next VPORT entry
VPORT
5
158D
330
151F
100
AcDbSymbolTableRecord
100
AcDbViewportTableRecord
2       <<< VPORT (configuration) name
*ACTIVE <<< same as first VPORT entry
70
0
10
0.0
20
0.0
11
0.45
21
0.5
12
2.01
22
-8.97
...
...
0
ENDTAB
```

LTYPE Table

The LTYPE table stores all line type definitions of a DXF drawing. Every line type used in the drawing has to have a table entry, or the DXF drawing is invalid for AutoCAD.

DXF R12 supports just simple line types, DXF R2000+ supports also complex line types with text or shapes included.

You have access to the line types table by the attribute *Drawing.linetypes*. The line type table itself is not stored in the entity database, but the table entries are stored in entity database, and can be accessed by its handle.

See also:

- DXF Reference: TABLES Section
- DXF Reference: LTYPE Table

Table Structure DXF R12

```
0           <<< start of table
TABLE
2           <<< set table type
LTYPE
70          <<< count of line types defined in this table, AutoCAD ignores this value
9
0           <<< 1. LTYPE table entry
LTYPE
            <<< LTYPE data tags
0           <<< 2. LTYPE table entry
LTYPE
            <<< LTYPE data tags and so on
0           <<< end of LTYPE table
ENDTAB
```

Table Structure DXF R2000+

```
0           <<< start of table
TABLE
2           <<< set table type
LTYPE
5           <<< LTYPE table handle
5F
330         <<< owner tag, tables has no owner
0
100         <<< subclass marker
AcDbSymbolTable
70          <<< count of line types defined in this table, AutoCAD ignores this value
9
0           <<< 1. LTYPE table entry
LTYPE
            <<< LTYPE data tags
0           <<< 2. LTYPE table entry
LTYPE
            <<< LTYPE data tags and so on
0           <<< end of LTYPE table
ENDTAB
```

Simple Line Type

ezdxf setup for line type 'CENTER':

```
dwg.linetypes.new("CENTER", dxfattribs={
    description = "Center ____ _ ____ _ ____ _ ____ _ ____ _ ____",
    pattern=[2.0, 1.25, -0.25, 0.25, -0.25],
 })
```

Simple Line Type Tag Structure DXF R2000+

```
0           <<< line type table entry
LTYPE
5           <<< handle of line type
1B1
330         <<< owner handle, handle of LTYPE table
5F
100         <<< subclass marker
AcDbSymbolTableRecord
100         <<< subclass marker
AcDbLinetypeTableRecord
2           <<< line type name
CENTER
70          <<< flags
0
3
Center ____ _ ____ _ ____ _ ____ _ ____ _ ____
72
65
73
4
40
2.0
49
1.25
74
0
49
-0.25
74
0
49
0.25
74
0
49
-0.25
74
0
```

Complex Line Type TEXT

ezdxf setup for line type 'GASLEITUNG':

```
dwg.linetypes.new('GASLEITUNG', dxfattribs={
    'description': 'Gasleitung2 ----GAS----GAS----GAS----GAS----GAS----GAS--',
    'length': 1,
    'pattern': 'A,.5,-.2,["GAS",STANDARD,S=.1,U=0.0,X=-0.1,Y=-.05],-.25',
})
```

TEXT Tag Structure

```
0
LTYPE
5
614
330
5F
100         <<< subclass marker
AcDbSymbolTableRecord
100         <<< subclass marker
AcDbLinetypeTableRecord
2
GASLEITUNG
70
0
3
Gasleitung2 ----GAS----GAS----GAS----GAS----GAS----GAS--
72
65
73
3
40
1
49
0.5
74
0
49
-0.2
74
2
75
0
340
11
46
0.1
50
0.0
44
-0.1
45
-0.05
9
GAS
49
-0.25
74
0
```

Complex Line Type SHAPE

ezdxf setup for line type 'GRENZE2':

```
dwg.linetypes.new('GRENZE2', dxfattribs={
    'description': 'Grenze eckig ----[]-----[]----[]-----[]----[]--',
    'length': 1.45,
    'pattern': 'A,.25,-.1,[132,ltypeshp.shx,x=-.1,s=.1],-.1,1',
})
```

SHAPE Tag Structure

```
0
LTYPE
5
615
330
5F
100         <<< subclass marker
AcDbSymbolTableRecord
100         <<< subclass marker
AcDbLinetypeTableRecord
2
GRENZE2
70
0
3
Grenze eckig ----[]-----[]----[]-----[]----[]--
72
65
73
4
40
1.45
49
0.25
74
0
49
-0.1
74
4
75
132
340
616
46
0.1
50
0.0
44
-0.1
45
0.0
49
-0.1
74
0
49
1.0
```

```
74
0
```

TODO

4.6.10 BLOCKS Section

The BLOCKS section contains all BLOCK definitions, beside the 'normal' reusable BLOCKS used by the INSERT entity, all layouts, as there are the model space and all paper space layouts, have at least a corresponding BLOCK definition in the BLOCKS section. The name of the model space BLOCK is `*Model_Space` (DXF R12: `$MODEL_SPACE`) and the name of the *active* paper space BLOCK is `*Paper_Space` (DXF R12: `$PAPER_SPACE`), the entities of these two layouts are stored in the ENTITIES section, the *inactive* paper space layouts are named by the scheme `*Paper_Spacennnn`, and the content of the inactive paper space layouts are stored in their BLOCK definition in the BLOCKS section.

The content entities of blocks are stored between the BLOCK and the ENDBKL entity.

BLOCKS section structure:

```
0           <<< start of a SECTION
SECTION
2           <<< start of BLOCKS section
BLOCKS
0           <<< start of 1. BLOCK definition
BLOCK
...         <<< Block content
...
0           <<< end of 1. Block definition
ENDBLK
0           <<< start of 2. BLOCK definition
BLOCK
...         <<< Block content
...
0           <<< end of 2. Block definition
ENDBLK
0           <<< end of BLOCKS section
ENDSEC
```

See also:

Block Management Structures Layout Management Structures

4.6.11 ENTITIES Section

TODO

4.6.12 OBJECTS Section

TODO

4.6.13 Data Model

Database Objects

(from the DXF Reference)

AutoCAD drawings consist largely of structured containers for database objects. Database objects each have the following features:

- A handle whose value is unique to the drawing/DXF file, and is constant for the lifetime of the drawing. This format has existed since AutoCAD Release 10, and as of AutoCAD Release 13, handles are always enabled.
- An optional xdata table, as entities have had since AutoCAD Release 11.
- An optional persistent reactor table.
- An optional ownership pointer to an extension dictionary which, in turn, owns subobjects placed in it by an application.

Symbol tables and symbol table records are database objects and, thus, have a handle. They can also have xdata and persistent reactors in their DXF records.

DXF R12 Data Model

The DXF R12 data model is identical to the file structure:

- HEADER section: common settings for the DXF drawing
- TABLES section: definitions for LAYERS, LINETYPE, STYLES ....
- BLOCKS section: block definitions and its content
- ENTITIES section: model space and paper space content

References are realized by simple names. The INSERT entity references the BLOCK definition by the BLOCK name, a TEXT entity defines the associated STYLE and LAYER by its name and so on, handles are not needed. Layout association of graphical entities in the ENTITIES section by the paper_space tag `(67, 0 or 1)`, 0 or missing tag means model space, 1 means paper space. The content of BLOCK definitions is enclosed by the BLOCK and the ENDBLK entity, no additional references are needed.

A clean and simple file structure and data model, which seems to be the reason why the DXF R12 Reference (released 1992) is still a widely used file format and Autodesk/AutoCAD supports the format by reading and writing DXF R12 files until today (DXF R13/R14 has no writing support by AutoCAD!).

TODO: list of available entities

See also:

More information about the DXF *DXF File Structure*

DXF R13+ Data Model

With the DXF R13 file format, handles are mandatory and they are really used for organizing the new data structures introduced with DXF R13.

The HEADER section is still the same with just more available settings.

The new CLASSES section contains AutoCAD specific data, has to be written like AutoCAD it does, but must not be understood.

The TABLES section got a new BLOCK_RECORD table - see *Block Management Structures* for more information.

The BLOCKS sections is mostly the same, but with handles, owner tags and new ENTITY types. Not active paper space layouts store their content also in the BLOCKS section - see *Layout Management Structures* for more information.

The ENTITIES section is also mostly same, but with handles, owner tags and new ENTITY types.

TODO: list of new available entities

And the new OBJECTS section - now its getting complicated!

Most information about the OBJECTS section is just guessed or gathered by trail and error (reverse engineering), because the documentation of the OBJECTS section and its objects in the DXF reference provided by Autodesk is very shallow. This is also the reason why I started the DXF Internals section, may be it helps other developers to start one or two steps above level zero.

The OBJECTS sections stores all the non-graphical entities of the DXF drawing. Non-graphical entities from now on just called 'objects' to differentiate them from graphical entities, just called 'entities'. The OBJECTS section follows commonly the ENTITIES section, but this is not mandatory. DXF R13 introduces also several new DXF objects, which resides exclusive in the OBJECTS section, taken from the DXF R14 reference, because I have no access to the DXF R13 reference, the DXF R13 reference is a compiled .hlp file which can't be read on Windows 10, a drastic real world example why it is better to avoid closed (proprietary) data formats ;):

- DICTIONARY: a general structural entity as a <name: handle> container
- ACDBDICTIONARYWDFLT: a DICTIONARY with a default value
- DICTIONARYVAR: used by AutoCAD to store named values in the database
- ACAD_PROXY_OBJECT: proxy object for entities created by other applications than AutoCAD
- GROUP: groups graphical entities without the need of a BLOCK definition
- IDBUFFER: just a list of references to objects
- IMAGEDEF: IMAGE definition structure, required by the IMAGE entity
- IMAGEDEF_REACTOR: also required by the IMAGE entity
- LAYER_INDEX: container for LAYER names
- MLINESTYLE
- OBJECT_PTR
- RASTERVARIABLES
- SPATIAL_INDEX: is always written out empty to a DXF file. This object can be ignored.
- SPATIAL_FILTER
- SORTENTSTABLE: control for regeneration/redraw order of entities
- XRECORD: used to store and manage arbitrary data. This object is similar in concept to XDATA but is not limited by size or order. Not supported by R13c0 through R13c3.

Still missing the LAYOUT object, which is mandatory in DXF R2000 to manage multiple paper space layouts. I don't know how DXF R13/R14 manages multiple layouts or if they even support this feature, but I don't care much about DXF R13/R14, because AutoCAD has no write support for this two formats anymore. ezdxf tries to upgrade this two DXF versions to DXF R2000 with the advantage of only two different data models to support: DXF R12 and DXF R2000+

New objects introduced by DXF R2000:

- LAYOUT: management object for model space and multiple paper space layouts
- ACDBPLACEHOLDER: surprise - just a place holder

New objects in DXF R2004:

- DIMASSOC
- LAYER_FILTER
- MATERIAL
- PLOTSETTINGS
- VBA_PROJECT

New objects in DXF R2007:

- DATATABLE
- FIELD
- LIGHTLIST
- RENDER
- RENDERENVIRONMENT
- MENTALRAYRENDERSETTINGS
- RENDERGLOBAL
- SECTION
- SUNSTUDY
- TABLESTYLE
- UNDERLAYDEFINITION
- VISUALSTYLE
- WIPEOUTVARIABLES

New objects in DXF R2013:

- GEODATA

New objects in DXF R2018:

- ACDBNAVISWORKSMODELDEF

Undocumented objects:

- SCALE
- ACDBSECTIONVIEWSTYLE

Objects Organisation

Many objects in the OBJECTS section are organized in a tree-like structure of DICTIONARY objects. Starting point for this data structure is the 'root' DICTIONARY with several entries to other DICTIONARY objects. The root DICTIONARY has to be the first object in the OBJECTS section. The management dicts for GROUP and LAYOUT objects are really important, but IMHO most of the other management tables are optional and for the most use cases not necessary. The ezdxf template for DXF R2018 contains only these entries in the root dict and most of them pointing to an empty DICTIONARY:

- ACAD_COLOR: points to an empty DICTIONARY
- **ACAD_GROUP:** supported by ezdxf
- **ACAD_LAYOUT:** supported by ezdxf

- ACAD_MATERIAL: points to an empty DICTIONARY
- ACAD_MLEADERSTYLE: points to an empty DICTIONARY
- ACAD_MLINESTYLE: points to an empty DICTIONARY
- ACAD_PLOTSETTINGS: points to an empty DICTIONARY
- **ACAD_PLOTSTYLENAME:** points to ACDBDICTIONARYWDFLT with one entry: 'Normal'
- ACAD_SCALELIST: points to an empty DICTIONARY
- ACAD_TABLESTYLE: points to an empty DICTIONARY
- ACAD_VISUALSTYLE: points to an empty DICTIONARY

Root DICTIONARY content for DXF R2018

```
0
SECTION
2       <<< start of the OBJECTS section
OBJECTS
0       <<< root DICTIONARY has to be the first object in the OBJECTS section
DICTIONARY
5       <<< handle
C
330     <<< owner tag
0       <<< always #0, has no owner
100
AcDbDictionary
281     <<< hard owner flag
1
3       <<< first entry
ACAD_CIP_PREVIOUS_PRODUCT_INFO
350     <<< handle to target (pointer)
78B     <<< points to a XRECORD with product info about the creator application
3       <<< entry with unknown meaning, if I shoul guess: something with about colors␣
↪...
ACAD_COLOR
350
4FB     <<< points to a DICTIONARY
3       <<< entry with unknown meaning
ACAD_DETAILVIEWSTYLE
350
7ED     <<< points to a DICTIONARY
3       <<< GROUP management, mandatory in all DXF versions
ACAD_GROUP
350
4FC     <<< points to a DICTIONARY
3       <<< LAYOUT management, mandatory if more than the *active* paper space is used
ACAD_LAYOUT
350
4FD     <<< points to a DICTIONARY
3       <<< MATERIAL management
ACAD_MATERIAL
350
4FE     <<< points to a DICTIONARY
3       <<< MLEADERSTYLE management
ACAD_MLEADERSTYLE
350
```

```
4FF     <<< points to a DICTIONARY
3       <<< MLINESTYLE management
ACAD_MLINESTYLE
350
500     <<< points to a DICTIONARY
3       <<< PLOTSETTINGS management
ACAD_PLOTSETTINGS
350
501     <<< points to a DICTIONARY
3       <<< plot style name management
ACAD_PLOTSTYLENAME
350
503     <<< points to a ACDBDICTIONARYWDFLT
3       <<< SCALE management
ACAD_SCALELIST
350
504     <<< points to a DICTIONARY
3       <<< entry with unknown meaning
ACAD_SECTIONVIEWSTYLE
350
7EB     <<< points to a DICTIONARY
3       <<< TABLESTYLE management
ACAD_TABLESTYLE
350
505     <<< points to a DICTIONARY
3       <<< VISUALSTYLE management
ACAD_VISUALSTYLE
350
506     <<< points to a DICTIONARY
3       <<< entry with unknown meaning
ACDB_RECOMPOSE_DATA
350
7F3
3       <<< entry with unknown meaning
AcDbVariableDictionary
350
7AE     <<< points to a DICTIONARY with handles to DICTIONARYVAR objects
0
DICTIONARY
...
...
0
ENDSEC
```

4.6.14 Block Management Structures

A BLOCK is a kind of layout like the model space or a paper space, with the similarity that all these layouts are containers for other graphical DXF entities. This block definition can be referenced in other layouts by the INSERT entity. By using block references the same set of graphical entities can be located multiple times at different layouts, this block references can be stretched and rotated without modifying the original entities. A block is referenced only by its name defined by the DXF tag `(2, name)`, there is a second DXF tag `(3, name2)` for the block name, which is not further documented by Autodesk, and I haven't tested what happens I the second name is different to the first block name.

The `(10, base_point)` tag (in BLOCK defines a insertion point of the block, by 'inserting' a block by the INSERT entity, this point of the block is placed at the location defined by the `(10, insert)` tag in the INSERT

entity, and it is also the base point for stretching and rotation.

A block defintion can contain INSERT entities, and it is possible to create cyclic block definitions (a BLOCK contains a INSERT of itself), but this should be avoided, CAD applications will not load the DXF file at all or maybe just crash. This is also the case for all other kinds of cyclic definitions like: BLOCK 'A' -> INSERT BLOCK 'B' and BLOCK 'B' -> INSERT BLOCK 'A'.

See also:

- ezdxf DXF Internals: *BLOCKS Section*
- DXF Reference: BLOCKS Section
- DXF Reference: BLOCK Entity
- DXF Reference: ENDBLK Entity
- DXF Reference: INSERT Entity

Block Names

Block names has to be unique and they are case insensitive ('Test' == 'TEST'). If there are two or more block definitions with the same name, AutoCAD (LT 2018) merges these blocks into a single block with unpredictable properties of all these blocks. In my test with two blocks, the final block has the name of the first block and the base-point of the second block, and contains all entities of both blocks.

Block Definitions in DXF R12

In DXF R12 the definition of a block is located in the BLOCKS section, no additional structures are needed. The definition starts with a BLOCK entity and ends with a ENDBLK entity. All entities between this two entities are the content of the block, the block is the owner of this entities like any layout.

As shown in the DXF file below (created by AutoCAD LT 2018), the BLOCK entity has no handle, but ezdxf writes also handles for the BLOCK entity and AutoCAD doesn't complain.

DXF R12 BLOCKS structure:

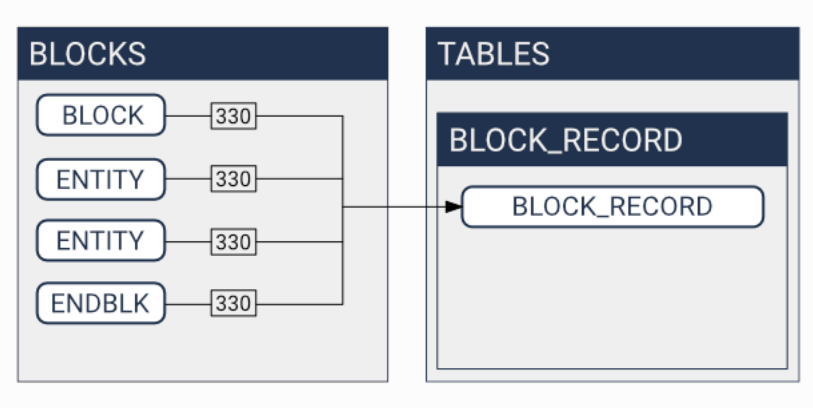
```
0           <<< start of a SECTION
SECTION
2           <<< start of BLOCKS section
BLOCKS
...         <<< model space and paper space block definitions not shown,
...         <<< see layout management
...
0           <<< start of a BLOCK definition
BLOCK
8           <<< layer, what this layer definition does is another fact, I don't know␣
↪(now)
0
2           <<< block name
ArchTick
70          <<< flags
1
10          <<< base point, x
0.0
20          <<< base point, y
0.0
30          <<< base point, z
0.0
```

```
3           <<< second BLOCK name, same as (2, name)
ArchTick
1           <<< xref name, if block is an external reference
            <<< empty string!
0           <<< start of the first entity of the BLOCK
LINE
5
28E
8
0
62
0
10
500.0
20
500.0
30
0.0
11
500.0
21
511.0
31
0.0
0           <<< start of the second entity of the BLOCK
LINE
...
0.0
0           <<< ENDBLK entity, marks the end of the BLOCK definition
ENDBLK
5           <<< ENDBLK gets a handle by AutoCAD, but BLOCK didn't
2F2
8           <<< as every entity, also ENDBLK requires a layer (same as BLOCK entity!)
0
0           <<< start of next BLOCK entity
BLOCK
...
0           <<< end BLOCK entity
ENDBLK
0           <<< end of BLOCKS section
ENDSEC
```

Block Definitions in DXF R2000 and later

The overall organization in the BLOCKS sections remains the same, but additional tags in the BLOCK entity, have to be maintained.

Especially the concept of ownership is important. Since DXF R13 every graphic entity is associated to a specific layout, and a BLOCK definition is a kind of layout. So all entities in the BLOCK definition, including the BLOCK and the ENDBLK entities, have an owner tag `(330, ...)`, which points to a BLOCK_RECORD entry in the BLOCK_RECORD table. As you can see in the chapter about *Layout Management Structures*, this concept is also valid for model space and paper space layouts, because these layouts are also BLOCKS, with the special difference, that entities of the model space and the *active* paper space are stored in the ENTITIES section.



See also:

- *Tag Structure DXF R13 and later*
- ezdxf DXF Internals: *TABLES Section*
- DXF Reference: TABLES Section
- DXF Reference: BLOCK_RECORD Entity

DXF R13 BLOCKS structure:

```
0           <<< start of a SECTION
SECTION
2           <<< start of BLOCKS section
BLOCKS
...         <<< model space and paper space block definitions not shown,
...         <<< see layout management
0           <<< start of BLOCK definition
BLOCK
5           <<< even BLOCK gets a handle now ;)
23A
330         <<< owner tag, the owner of a BLOCK is a BLOCK_RECORD in the BLOCK_RECORD␣
↪table
238
100         <<< subclass marker
AcDbEntity
8           <<< layer of the BLOCK definition
0
100         <<< subclass marker
AcDbBlockBegin
2           <<< BLOCK name
ArchTick
70          <<< flags
0
10          <<< base point, x
0.0
20          <<< base point, y
0.0
30          <<< base point, z
0.0
3           <<< second BLOCK name, same as (2, name)
ArchTick
1           <<< xref name, if block is an external reference
            <<< empty string!
0           <<< start of the first entity of the BLOCK
```

```
LWPOLYLINE
5
239
330         <<< owner tag of LWPOLYLINE
238         <<< handle of the BLOCK_RECORD!
100
AcDbEntity
8
0
6
ByBlock
62
0
100
AcDbPolyline
90
2
70
0
43
0.15
10
-0.5
20
-0.5
10
0.5
20
0.5
0           <<< ENDBLK entity, marks the end of the BLOCK definition
ENDBLK
5           <<< handle
23B
330         <<< owner tag, same BLOCK_RECORD as for the BLOCK entity
238
100         <<< subclass marker
AcDbEntity
8           <<< as every entity, also ENDBLK requires a layer (same as BLOCK entity!)
0
100         <<< subclass marker
AcDbBlockEnd
0           <<< start of the next BLOCK
BLOCK
...
0
ENDBLK
...
0           <<< end of the BLOCKS section
ENDSEC
```

DXF R13 BLOCK_RECORD structure:

```
0           <<< start of a SECTION
SECTION
2           <<< start of TABLES section
TABLES
0           <<< start of a TABLE
TABLE
```

```
2           <<< start of the BLOCK_RECORD table
BLOCK_RECORD
5           <<< handle of the table (INFO: ezdxf doesn't store tables in the entities␣
↪database)
1
330         <<< owner tag of the table
0           <<< is always #0
100         <<< subclass marker
AcDbSymbolTable
70          <<< count of table entries, not reliable
4
0           <<< start of first BLOCK_RECORD entry
BLOCK_RECORD
5           <<< handle of BLOCK_RECORD, in ezdxf often refered as 'layout key'
1F
330         <<< owner of the BLOCK_RECORD is the BLOCK_RECORD table
1
100         <<< subclass marker
AcDbSymbolTableRecord
100         <<< subclass marker
AcDbBlockTableRecord
2           <<< name of the BLOCK or LAYOUT
*Model_Space
340         <<< pointer to the associated LAYOUT object
4AF
70          <<< AC1021 (R2007) block insertion units
0
280         <<< AC1021 (R2007) block explodability
1
281         <<< AC1021 (R2007) block scalability
0

...         <<< paper space not shown
...
0           <<< next BLOCK_RECORD
BLOCK_RECORD
5           <<< handle of BLOCK_RECORD, in ezdxf often refered as 'layout key'
238
330         <<< owner of the BLOCK_RECORD is the BLOCK_RECORD table
1
100         <<< subclass marker
AcDbSymbolTableRecord
100         <<< subclass marker
AcDbBlockTableRecord
2           <<< name of the BLOCK
ArchTick
340         <<< pointer to the associated LAYOUT object
0           <<< #0, because BLOCK doesn't have an associated LAYOUT object
70          <<< AC1021 (R2007) block insertion units
0
280         <<< AC1021 (R2007) block explodability
1
281         <<< AC1021 (R2007) block scalability
0
0           <<< end of BLOCK_RECORD table
ENDTAB
0           <<< next TABLE
TABLE
```

```
...
0
ENDTAB
0           <<< end of TABLES section
ENDESC
```

4.6.15 Layout Management Structures

Layouts are separated entity spaces, there are three different Layout types:

1. Model space contains the 'real' world representation of the drawing subject in real world units.
2. Paper space are used to create different drawing sheets of the subject for printing or PDF export
3. Blocks are reusable sets of graphical entities, inserted by the INSERT entity.

All layouts have at least a BLOCK definition in the BLOCKS section and since DXF R13 exists the BLOCK_RECORD table with an entry for every BLOCK in the BLOCKS section.

See also:

Information about *Block Management Structures*

The name of the model space BLOCK is `*Model_Space` (DXF R12: `$MODEL_SPACE`) and the name of the *active* paper space BLOCK is `*Paper_Space` (DXF R12: `$PAPER_SPACE`), the entities of these two layouts are stored in the ENTITIES section, DXF R12 supports just one paper space layout.

DXF R13 and later supports multiple paper space layouts, the *active* layout is still called `*Paper_Space`, the additional *inactive* paper space layouts are named by the scheme `*Paper_Spacennnn`, where the first inactive paper space is called `*Paper_Space0`, the second `*Paper_Space1` and so on. A none consecutive numbering is tolerated by AutoCAD. The content of the inactive paper space layouts are stored as BLOCK content in the BLOCKS section. These names are just the DXF internal layout names, each layout has an additional layout name which is displayed to the user by the CAD application.

A BLOCK definition and a BLOCK_RECORD is not enough for a proper layout setup, an LAYOUT entity in the OBJECTS section is also required. All LAYOUT entities are managed by a DICTIONARY entity, which is referenced as `ACAD_LAYOUT` entity in the root DICTIONARY of the DXF file.

Note: All floating point values are rounded to 2 decimal places for better readability.

LAYOUT Entiy

Since DXF R2000 model space and paper space layouts require the DXF LAYOUT entity.

```
0
LAYOUT
5       <<< handle
59
102     <<< extension dictionary (ignore)
{ACAD_XDICTIONARY
360
1C3
102
}
102     <<< reactor (required?)
{ACAD_REACTORS
```

```
330
1A      <<< pointer to "ACAD_LAYOUT" DICTIONARY (layout management table)
102
}
330     <<< owner handle
1A      <<< pointer to "ACAD_LAYOUT" DICTIONARY (same as reactor pointer)
100     <<< PLOTSETTINGS
AcDbPlotSettings
1       <<< page setup name

2       <<< name of system printer or plot configuration file
none_device
4       <<< paper size, part in braces should follow the schema (width_x_height_unit)␣
↪unit is 'Inches' or 'MM'
Letter_(8.50_x_11.00_Inches)  # the part in front of the braces is ignored by AutoCAD
6       <<< plot view name

40      <<< size of unprintable margin on left side of paper in millimeters, defines␣
↪also the plot origin-x
6.35
41      <<< size of unprintable margin on bottom of paper in millimeters, defines␣
↪also the plot origin-y
6.35
42      <<< size of unprintable margin on right side of paper in millimeters
6.35
43      <<< size of unprintable margin on top of paper in millimeters
6.35
44      <<< plot paper size: physical paper width in millimeters
215.90
45      <<< plot paper size: physical paper height in millimeters
279.40
46      <<< X value of plot origin offset in millimeters, moves the plot origin-x
0.0
47      <<< Y value of plot origin offset in millimeters, moves the plot origin-y
0.0
48      <<< plot window area: X value of lower-left window corner
0.0
49      <<< plot window area: Y value of lower-left window corner
0.0
140     <<< plot window area: X value of upper-right window corner
0.0
141     <<< plot window area: Y value of upper-right window corner
0.0
142     <<< numerator of custom print scale: real world (paper) units, 1.0 for scale␣
↪1:50
1.0
143     <<< denominator of custom print scale: drawing units, 50.0 for scale 1:50
1.0
70      <<< plot layout flags, bit-coded (... too many options)
688     <<< b1010110000 = UseStandardScale(16)/PlotPlotStyle(32)/
↪PrintLineweights(128)/DrawViewportsFirst(512)
72      <<< plot paper units (0/1/2 for inches/millimeters/pixels), are pixels really␣
↪supported?
0
73      <<< plot rotation (0/1/2/3 for 0deg/90deg counter-cw/upside-down/90deg cw)
1       <<< 90deg clockwise
74      <<< plot type 0-5 (... too many options)
5       <<< 5 = layout information
```

```
7       <<< current plot style name, e.g. 'acad.ctb' or 'acadlt.ctb'

75      <<< standard scale type 0-31 (... too many options)
16      <<< 16 = 1:1, also 16 if user scale type is used
147     <<< unit conversion factor
1.0     <<< for plot paper units in mm, else  0.03937... (1/25.4) for inches as plot␣
↪paper units
76      <<< shade plot mode (0/1/2/3 for as displayed/wireframe/hidden/rendered)
0       <<< as displayed
77      <<< shade plot resolution level 1-5 (... too many options)
2       <<< normal
78      <<< shade plot custom DPI: 100-32767, Only applied when shade plot resolution␣
↪level is set to 5 (Custom)
300
148     <<< paper image origin: X value
0.0
149     <<< paper image origin: Y value
0.0
100     <<< LAYOUT settings
AcDbLayout
1       <<< layout name
Layout1
70      <<< flags bit-coded
1       <<< 1 = Indicates the PSLTSCALE value for this layout when this layout is␣
↪current
71      <<< Tab order ("Model" tab always appears as the first tab regardless of its␣
↪tab order)
1
10      <<< minimum limits for this layout (defined by LIMMIN while this layout is␣
↪current)
-0.25   <<<     x value, distance of the left paper margin from the plot origin-x, in␣
↪plot paper units and by scale (e.g. x50 for 1:50)
20      <<<     group code for y value
-0.25   <<<     y value, distance of the bottom paper margin from the plot origin-y,␣
↪in plot paper units and by scale (e.g. x50 for 1:50)
11      <<< maximum limits for this layout (defined by LIMMAX while this layout is␣
↪current)
10.75   <<<     x value, distance of the right paper margin from the plot origin-x,␣
↪in plot paper units and by scale (e.g. x50 for 1:50)
21      <<<     group code for y value
8.25    <<<     y value, distance of the top paper margin from the plot origin-y, in␣
↪plot paper units and by scale (e.g. x50 for 1:50)
12      <<< insertion base point for this layout (defined by INSBASE while this␣
↪layout is current)
0.0     <<<     x value
22      <<<     group code for y value
0.0     <<<     y value
32      <<<     group code for z value
0.0     <<<     z value
14      <<< minimum extents for this layout (defined by EXTMIN while this layout is␣
↪current), AutoCAD default is (1e20, 1e20, 1e20)
1.05    <<<     x value
24      <<<     group code for y value
0.80    <<<     y value
34      <<<     group code for z value
0.0     <<<     z value
15      <<< maximum extents for this layout (defined by EXTMAX while this layout is␣
↪current), AutoCAD default is (-1e20, -1e20, -1e20)
```

```
9.45    <<<     x value
25      <<<     group code for y value
7.20    <<<     y value
35      <<<     group code for z value
0.0     <<<     z value
146     <<< elevation ???
0.0
13      <<< UCS origin (3D Point)
0.0     <<<     x value
23      <<<     group code for y value
0.0     <<<     y value
33      <<<     group code for z value
0.0     <<<     z value
16      <<< UCS X-axis (3D vector)
1.0     <<<     x value
26      <<<     group code for y value
0.0     <<<     y value
36      <<<     group code for z value
0.0     <<<     z value
17      <<< UCS Y-axis (3D vector)
0.0     <<<     x value
27      <<<     group code for y value
1.0     <<<     y value
37      <<<     group code for z value
0.0     <<<     z value
76      <<< orthographic type of UCS 0-6 (... too many options)
0       <<< 0 = UCS is not orthographic ???
330     <<< ID/handle of required block table record
58
331     <<< ID/handle to the viewport that was last active in this layout when the␣
↪layout was current
1B9
1001    <<< extended data (ignore)
...
```

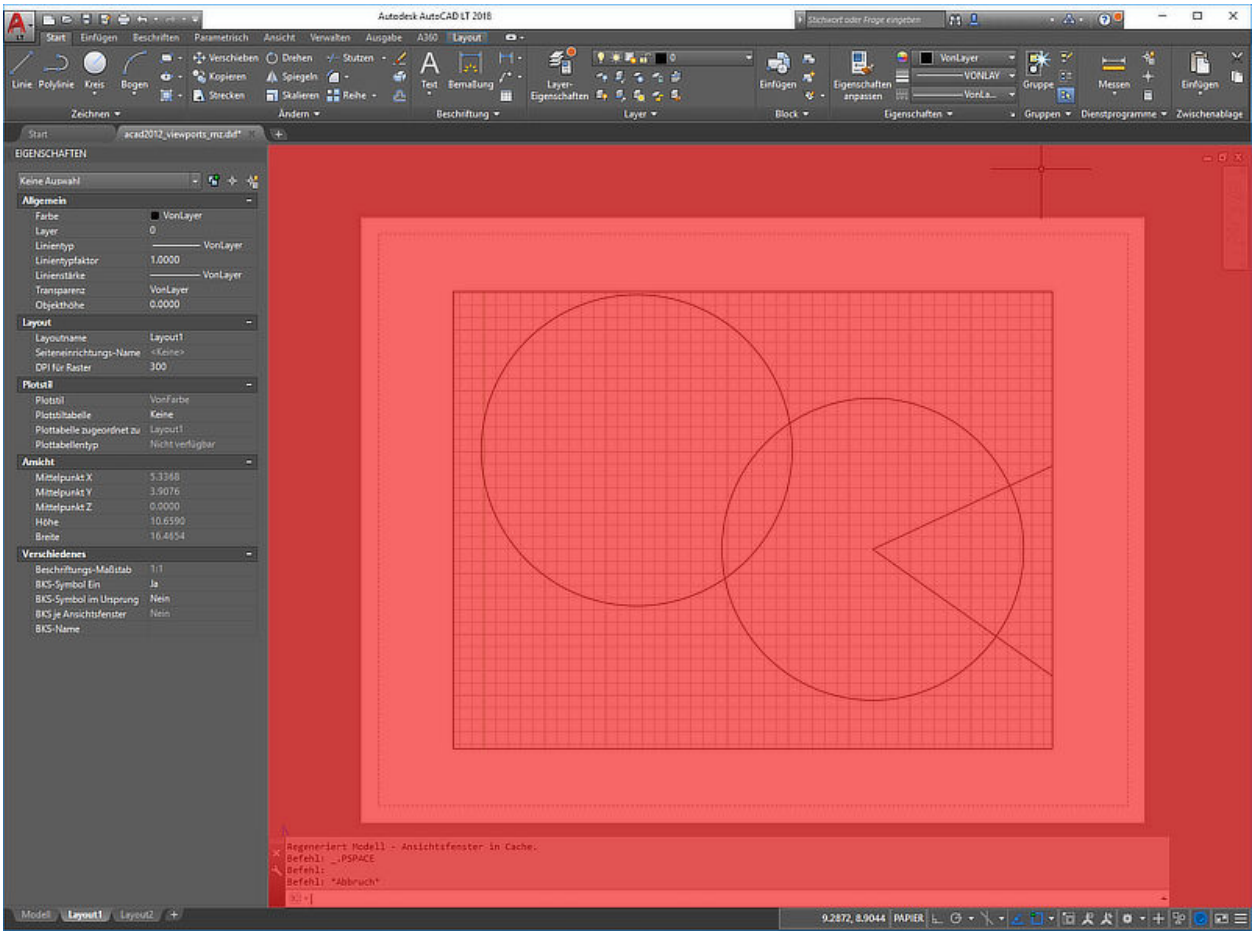
And as it seems this is also not enough for a well defined LAYOUT, at least a "main" VIEWPORT entity with ID=1 is required for paper space layouts, located in the entity space of the layout.

The model space layout requires (?) a VPORT entity in the VPORT table (group code 331 in the AcDbLayout subclass).

Main VIEWPORT Entity for LAYOUT

The "main" viewport for layout `Layout1` shown above. This viewport is located in the associated BLOCK definition called `*Paper_Space0`. Group code 330 in subclass AcDbLayout points to the BLOCK_RECORD of `*Paper_Space0`. Remember: the entities of the *active* paper space layout are located in the ENTITIES section, therefor `Layout1` is not the active paper space layout.

The "main" VIEWPORT describes, how the application shows the paper space layout on the screen, and I guess only AutoCAD needs this values. And the most values



```
0
VIEWPORT
5       <<< handle
1B4
102     <<< extension dictionary (ignore)
{ACAD_XDICTIONARY
360
1B5
102
}
330     <<< owner handle
58      <<< points to BLOCK_RECORD (same as group code 330 in AcDbLayout of "Layout1")
100
AcDbEntity
67      <<< paper space flag
1       <<< 0 = model space; 1 = paper space
8       <<< layer,
0
100
AcDbViewport
10      <<< Center point (in WCS)
5.25    <<<     x value
20      <<<     group code for y value
4.00    <<<     y value
30      <<<     group code for z value
0.0     <<<     z value
```

```
40       <<< width in paper space units
23.55    <<< VIEW size in AutoCAD, depends on the workstation configuration
41       <<< height in paper space units
9.00     <<< VIEW size in AutoCAD, depends on the workstation configuration
68       <<< viewport status field -1/0/n
2        <<< >0  On and active. The value indicates the order of stacking for the␣
↪viewports, where 1 is the active viewport, 2 is the next, and so forth
69       <<< viewport ID
1        <<< "main" viewport has always ID=1
12       <<< view center point in Drawing Coordinate System (DCS), defines the center␣
↪point of the VIEW in relation to the LAYOUT origin
5.25     <<<     x value
22       <<<     group code for y value
4.00     <<<     y value
13       <<< snap base point in model space
0.0      <<<     x value
23       <<<     group code for y value
0.0      <<<     y value
14       <<< snap spacing in model space units
0.5      <<<     x value
24       <<<     group code for y value
0.5      <<<     y value
15       <<< grid spacing in model space units
0.5      <<<     x value
25       <<<     group code for y value
0.5      <<<     y value
16       <<< view direction vector from target (in WCS)
0.0      <<<     x value
26       <<<     group code for y value
0.0      <<<     y value
36       <<<     group code for z value
1.0      <<<     z value
17       <<< view target point
0.0      <<<     x value
27       <<<     group code for y value
0.0      <<<     y value
37       <<<     group code for z value
0.0      <<<     z value
42       <<<     perspective lens length, focal length?
50.0     <<<     50mm
43       <<<     front clip plane z value
0.0      <<<     z value
44       <<<     back clip plane z value
0.0      <<<     z value
45       <<<     view height (in model space units)
9.00
50       <<< snap angle
0.0
51       <<< view twist angle
0.0
72       <<< circle zoom percent
1000
90       <<< Viewport status bit-coded flags (... too many options)
819232   <<< b1100100000000000100000
1        <<< plot style sheet name assigned to this viewport

281      <<< render mode (... too many options)
0        <<< 0 = 2D optimized (classic 2D)
```

```
71      <<< UCS per viewport flag
1       <<< 1 = This viewport stores its own UCS which will become the current UCS␣
↪whenever the viewport is activated
74      <<< Display UCS icon at UCS origin flag
0       <<< this field is currently being ignored and the icon always represents the␣
↪viewport UCS
110     <<< UCS origin (3D point)
0.0     <<<     x value
120     <<<     group code for y value
0.0     <<<     y value
130     <<<     group code for z value
0.0     <<<     z value
111     <<< UCS X-axis (3D vector)
1.0     <<<     x value
121     <<<     group code for y value
0.0     <<<     y value
131     <<<     group code for z value
0.0     <<<     z value
112     <<< UCS Y-axis (3D vector)
0.0     <<<     x value
122     <<<     group code for y value
1.0     <<<     y value
132     <<<     group code for z value
0.0     <<<     z value
79      <<< Orthographic type of UCS (... too many options)
0       <<< 0 = UCS is not orthographic
146     <<< elevation
0.0
170     <<< shade plot mode (0/1/2/3 for as displayed/wireframe/hidden/rendered)
0       <<< as displayed
61      <<< frequency of major grid lines compared to minor grid lines
5       <<< major grid subdivided by 5
348     <<< visual style ID/handle (optional)
9F
292     <<< default lighting flag, on when no user lights are specified.
1
282     <<< Default lighting type (0/1 = one distant light/two distant lights)
1       <<< one distant light
141     <<< view brightness
0.0
142     <<< view contrast
0.0
63      <<< ambient light color (ACI), write only if not black color
250
421     <<< ambient light color (RGB), write only if not black color
3355443
```

CHAPTER 5

News

Version 0.8.7 - 2018-03-04

- Release notes: https://ezdxf.mozman.at/release-v0-8-7.html
- NEW: entity.get_layout() returns layout in which entity resides or None if unassigned
- NEW: copy any DXF entity by entity.copy() without associated layout, add copy to any layout you want, by layout.add_entity().
- NEW: copy entity to another layout by entity.copy_to_layout(layout)
- NEW: move entity from actual layout to another layout by entity.move_to_layout(layout)
- NEW: support for splines by control points: add_open_spline(), add_closed_spline(), add_rational_spline(), add_closed_rational_spline()
- NEW: bspline_control_frame() calculates B-spline control points from fit points, but not the same as AutoCAD
- NEW: R12Spline add-on, 2d B-spline with control frame support by AutoCAD, but curve is just an approximated POLYLINE
- NEW: added entity.get_flag_state() and entity.set_flag_state() for easy access to binary coded flags
- NEW: set new $FINGERPRINTGUID for new drawings
- NEW: set new $VERSIONGUID on saving a drawing
- NEW: improved IMAGE support, by adding RASTERVARIABLES entity, use Drawing.set_raster_variables(frame, quality, units)
- BUGFIX: closing user defined image boundary path automatically, else AutoCAD crashes

Version 0.8.6 - 2018-02-17

- Release notes: https://ezdxf.mozman.at/release-v0-8-6.html
- NEW: ezdxf project website: https://ezdxf.mozman.at/
- CHANGE: create all missing tables of the TABLES sections for DXF R12
- BUGFIX: entities on new layouts will be saved

- NEW: Layout.page_setup() and correct 'main' viewport for DXF R2000+; For DXF R12 page_setup() exists, but does not provide useful results. Page setup for DXF R12 is still a mystery to me.
- NEW: Table(), MText(), Ellipse(), Spline(), Bezier(), Clothoid(), LinearDimension(), RadialDimension(), ArcDimension() and AngularDimension() composite objects from dxfwrite as addons, these addons support DXF R12
- NEW: geometry builder as addons: MeshBuilder(), MeshVertexMerger(), MengerSponge(), SierpinskyPyramid(), these addons require DXF R2000+ (MESH entity)
- BUGFIX: fixed invalid implementation of context manager for r12writer

Version 0.8.5 - 2018-01-28

- Release notes: https://ezdxf.mozman.at/release-v0-8-5.html
- CHANGE: block names are case insensitive 'TEST' == 'Test' (like AutoCAD)
- CHANGE: table entry (layer, linetype, style, dimstyle, ...) names are case insensitive 'TEST' == 'Test' (like AutoCAD)
- CHANGE: raises DXFInvalidLayerName() for invalid characters in layer names: <>/":;?*|='
- CHANGE: audit process rewritten
- CHANGE: skip all comments, group code 999
- CHANGE: removed compression for unused sections (THUMBNAILSECTION, ACDSDATA)
- NEW: write DXF R12 files without handles: set dwg.header['$HANDLING']=0, default value is 1
- added subclass marker filter for R12 and prior files in legacy_mode=True (required for malformed DXF files)
- removed special check for Leica Disto Unit files, use readfile(filename, legacy_mode=True) (malformed DXF R12 file, see previous point)

Version 0.8.4 - 2018-01-14

- Release notes: https://ezdxf.mozman.at/release-v0-8-4.html
- NEW: Support for complex line types with text or shapes
- NEW: DXF file structure validator at SECTION level, tags outside of sections will be removed
- NEW: Basic read support for DIMENSION
- CHANGE: improved exception management, in the future ezdxf should only raise exceptions inherited from DXFError for DXF related errors, previous exception classes still work
  - DXFValueError(DXFError, ValueError)
  - DXFKeyError(DXFError, KeyError)
  - DXFAttributeError(DXFError, AttributeError)
  - DXFIndexError(DXFError, IndexError)
  - DXFTableEntryError(DXFValueError)
- speedup low level tag reader around 5%, and speedup tag compiler around 5%

Version 0.8.3 - 2018-01-02

- CHANGE: Lwpolyline - suppress yielding z coordinates if they exists (DXFStructureError: z coordinates are not defined in the DXF standard)
- NEW: setup creates a script called 'dxfpp' (DXF Pretty Printer) in the Python script folder
- NEW: basic support for DXF format AC1032 introduced by AutoCAD 2018

- NEW: ezdxf use logging and writes all logs to a logger called 'ezdxf'. Logging setup is the domain of the application!
- NEW: warns about multiple block definitions with the same name in a DXF file. (DXFStructureError)
- NEW: legacy_mode parameter in ezdxf.read() and ezdxf.readfile(): tries do fix coordinate order in LINE entities (10, 11, 20, 21) by the cost of around 5% overall speed penalty at DXF file loading

Version 0.8.2 - 2017-05-01

- NEW: Insert.delete_attrib(tag) - delete ATTRIB entities from the INSERT entity
- NEW: Insert.delete_all_attribs() - delete all ATTRIB entities from the INSERT entity
- BUGFIX: setting attribs_follow=1 at INSERT entity before adding an attribute entity works

Version 0.8.1 - 2017-04-06

- NEW: added support for constant ATTRIB/ATTDEF to the INSERT (block reference) entity
- NEW: added ATTDEF management methods to BlockLayout (has_attdef, get_attdef, get_attdef_text)
- NEW: added (read/write) properties to ATTDEF/ATTRIB for setting flags (is_const, is_invisible, is_verify, is_preset)

Version 0.8.0 - 2017-03-28

- added groupby(dxfattrib='', key=None) entity query function, it is supported by all layouts and the query result container: Returns a dict, where entities are grouped by a dxfattrib or the result of a key function.
- added ezdxf.audit() for DXF error checking for drawings created by ezdxf - but not very capable yet
- dxfattribs in factory functions like add_line(dxfattribs=...), now are copied internally and stay unchanged, so they can be reused multiple times without getting modified by ezdxf.
- removed deprecated Drawing.create_layout() -> Drawing.new_layout()
- removed deprecated Layouts.create() -> Layout.new()
- removed deprecated Table.create() -> Table.new()
- removed deprecated DXFGroupTable.add() -> DXFGroupTable.new()
- BUFIX in EntityQuery.extend()

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

5.1 Indices and tables

- genindex
- search

Python Module Index

e

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z