dxfwrite Documentation

Release 1.1.0

Manfred Moitzi

Apr 23, 2017

Contents

Welcome! This is the documentation for dxfwrite 1.1.0, last updated Apr 23, 2017.

A Python library to create DXF R12 drawings.

usage:

```
from dxfwrite import DXFEngine as dxf

drawing = dxf.drawing('test.dxf')
drawing.add_layer('LINES')
drawing.add(dxf.line((0, 0), (1, 0), color=7, layer='LINES'))
drawing.save()
```

First create a *Drawing*, then create various drawing entities by `DXFEngine` and add them to the drawing with the `Drawing.add()` method. *Layers*, *Textstyles*, *Linetypes*, *Views* and *Viewports* were also created by the `DXFEngine` factory.

CHAPTER 1

Contents

Drawing

class `Drawing`

The Drawing object manages all the necessary sections, like header, tables and blocks. The tables-attribute contains the layers, styles, linetypes and other tables.

`Drawing.__init__`(*name='noname.dxf'*)

Parameters **name** (*str*) – filename of drawing

Methods

`Drawing.add`(*entity*)

Add an entity to drawing.

shortcut for: Drawing.entities.add()

`Drawing.save`()

Write DXF data to file-system.

`Drawing.saveas`(*name*)

Set new filename and write DXF data to file-system.

`Drawing.add_layer`(*name*, ***kwargs*)

Define a new layer. For valid keyword args see: *LAYER*

`Drawing.add_style`(*name*, ***kwargs*)

Define a new text-style. For valid keyword args see: *TEXTSTYLE*

`Drawing.add_linetype`(*name*, ***kwargs*)

Define a new linetype. For valid keyword args see: *LINETYPE*

`Drawing.add_view`(*name*, ***kwargs*)

Define a new view. For valid keyword args see: *VIEW*

Drawing.**add_viewport**(*name*, ***kwargs*)
Define a new viewport. For valid keyword args see: *VIEWPORT (Table Entry)*

Drawing.**add_xref**(*filepath*, *insert=(0., 0., 0.)*, *layer='0'*)
Create a simple XREF reference, *filepath* is the referenced drawing and *insert* is the insertion point.

Attributes

header
the header section, see *HEADER Section*

modelspace
Provides only a *add* method for adding entities to the *modelspace*, does the same as the `add()` method of the *drawing* object, except it garantees the *paper_space* attribute of the added entity is `'0'`.

paperspace
Provides only a *add* method for adding entities to the *paperspace*, does the same as the `add()` method of the *drawing* object, except it garantees the *paper_space* attribute of the added entity is `'1'`.

> **Warning:** DXF R12 supports only **one** paperspace.

usage:

```
from dxfwrite import DXFEngine as dxf

drawing = dxf.drawing(name='test.dxf')
drawing.paperspace.add(dxf.text('Text in paperspace'))
drawing.modelspace.add(dxf.text('Text in modelspace'))
drawing.add(dxf.text('Text also in paperspace', insert=(0, 1), paper_space=1))
drawing.add(dxf.text('Text also in modelspace', insert=(0, 1)))
```

blocks
the blocks section, see *BLOCK* definition.

usage:

```
from dxfwrite import DXFEngine as dxf

drawing = dxf.drawing(name='test.dxf')
drawing.add_layer('LINES')
drawing.add(dxf.line((0, 0), (10, 0), layer='LINES')))

# set header vars, see dxf documentation for header var explanation.
# set string
drawing.header['$CLAYER'] = 'CurrentLayer'

# set int/float
drawing.header['$ANGBASE'] = 30

# set 3D Point
drawing.header['$EXTMIN'] = (0, 0, -10)
drawing.header['$EXTMAX'] = (100, 100, 50)

# add a block definition to the drawing
drawing.blocks.add(blockdef)
```

DXFEngine

DXFEngine is the dxf entity creation factory, the dxfwrite interface.

class DXFEngine
Factory, creates the dxf objects.

This is the dedicated interface to dxfwrite, all table entries and all all DXF entities should be created by the methods of this object. All methods are static methods, so this object hasn't to be instantiated.

Drawing

DXFEngine.**drawing**(*name='empty.dxf'*)
Create a new drawing.

The drawing-object contains all the sections, tables and entities, which are necessary for a valid dxf-drawing.

For drawing methods see *Drawing* class.

Table Entries

DXFEngine.**layer**(*name*, ***kwargs*)
Create a new layer.

Parameters

- **name** (*str*) – layer name
- **flags** (*int*) – standard flag values, bit-coded, default=0
- **color** (*int*) – color number, negative if layer is off, default=1
- **linetype** (*str*) – linetype name, default="CONTINUOUS"

See also:

LAYER

DXFEngine.**style**(*name*, ***kwargs*)
Create a new textstyle.

Parameters

- **name** (*str*) – textstyle name
- **flags** (*int*) – standard flag values, bit-coded, default=0
- **generation_flags** (*int*) – text generation flags, default = 0
- **height** (*float*) – fixed text height, 0 if not fixed = default
- **last_height** – last height used, default=1.
- **width** (*float*) – width factor, default=1.
- **oblique** (*float*) – oblique angle in degree, default=0.
- **font** (*str*) – primary font filename, default="ARIAL"
- **bigfont** (*str*) – big-font file name, default=""

See also:

TEXTSTYLE

DXFEngine.**linetype**(*name*, ***kwargs*)
Create a new linetype.

Parameters

- **name** (*str*) – linetype name
- **flags** (*int*) – standard flag values, bit-coded, default=0
- **description** (*str*) – descriptive text for linetype, default=""
- **pattern** – line pattern definition, see method *DXFEngine.linepattern*

See also:

LINETYPE

DXFEngine.**linepattern**(*pattern*)
Create a *LINEPATTERN* object from pattern-list.

example linepattern([2.0, 1.25, -0.25, 0.25, -0.25]), for format description see object *LINEPATTERN*.

DXFEngine.**view**(*name*, ***kwargs*)
Create a new view.

Parameters

- **name** (*str*) – view name
- **flags** (*int*) – standard flag values, bit-coded, default=0 STD_FLAGS_PAPER_SPACE, if set this is a paper space view.
- **height, width** (*float*) – view height and width, in DCS?!, default=1.0
- **center_point** – view center point, in DCS?! (xy-tuple), default=(.5, .5)
- **direction_point** – view direction from target point, in WCS!! (xyz-tuple), default=(0, 0, 1)
- **target_point** – target point, in WCS!! (xyz-tuple), default=(0, 0, 0)
- **lens_length** (*float*) – lens length, default=50
- **front_clipping** (*float*) – front and back clipping planes, offsets from target point, default=0
- **back_clipping** – see front_clipping
- **view_twist** (*float*) – twist angle in degree, default=0
- **view_mode** (*int*) – view mode, bit-coded, default=0

DXFEngine.**vport**(*name*, ***kwargs*)
Create a new viewport table entry.

Parameters

- **name** (*str*) – viewport name
- **flags** (*int*) – standard flag values, bit-coded, default=0
- **lower_left** – lower-left corner of viewport, (xy-tuple), default=(0, 0)
- **upper_right** – upper-right corner of viewport, (xy-tuple), default=(1, 1)
- **center_point** – view center point, in WCS, (xy-tuple), default=(.5, .5)
- **snap_base** – snap base point, (xy-tuple), default=(0, 0)

- **snap_spacing** – snap spacing, X and Y (xy-tuple), default=(.1, .1)
- **grid_spacing** – grid spacing, X and Y (xy-tuple), default=(.1, .1)
- **direction_point** – view from direction point to target point (xyz-tuple), default=(0, 0, 1)
- **target_point** – view target point (xyz-tuple), default=(0, 0, 0)
- **aspect_ratio** – viewport aspect ratio (float), default=1.
- **lens_length** (*float*) – lens length, default=50
- **front_clipping** (*float*) – front and back clipping planes, offsets from target point , default=0
- **back_clipping** (*float*) – see front_clipping
- **view_twist** (*float*) – twist angle in degree, default=0
- **circle_zoom** (*float*) – circle zoom percent, default=100
- **view_mode** (*int*) – view mode, bit-coded, default=0
- **fast_zoom** (*int*) – fast zoom setting, default=1
- **ucs_icon** (*int*) – UCSICON settings, default=3
- **snap_on** (*int*) – snap on/off, default=0
- **grid_on** (*int*) – grid on/off, default=0
- **snap_style** (*int*) – snap style, default=0
- **snap_isopair** (*int*) – snap isopair, default=0

viewmode flags for *view* and *viewport*:

- VMODE_TURNED_OFF
- VMODE_PERSPECTIVE_VIEW_ACTIVE
- VMODE_FRONT_CLIPPING_ON
- VMODE_BACK_CLIPPING_ON
- VMODE_UCS_FOLLOW_MODE_ON
- VMODE_FRONT_CLIP_NOT_AT_EYE

DXFEngine.**ucs**(*name*, ***kwargs*)

Create a new user-coordinate-system (UCS).

Parameters

- **name** (*str*) – ucs name
- **flags** (*int*) – standard flag values, bit-coded
- **origin** – origin in WCS (xyz-tuple), default=(0, 0, 0)
- **xaxis** – xaxis direction in WCS (xyz-tuple), default=(1, 0, 0)
- **yaxis** – yaxis direction in WCS (xyz-tuple), default=(0, 1, 0)

DXFEngine.**appid**(*name*)

DXF R12 Entities

DXFEngine.**arc**(*radius=1.0*, *center=(0., 0.)*, *startangle=0.*, *endangle=360.*, ***kwargs*)
Create a new arc-entity.

Parameters

- **radius** (*float*) – arc radius
- **center** – center point (xy- or xyz-tuple), z-axis is 0 by default
- **startangle** (*float*) – start angle in degree
- **endangle** (*float*) – end angle in degree

See also:

ARC

DXFEngine.**attdef**(*tag*, *insert=(0., 0.)*, ***kwargs*)
Create a new attribute definition, used in block-definitions.

Parameters

- **text** (*str*) – attribute default text
- **insert** – insert point (xy- or xyz-tuple), z-axis is 0 by default
- **prompt** (*str*) – prompt text, like "insert a value:"
- **tag** (*str*) – attribute tag string
- **flags** (*int*) – attribute flags, bit-coded, default=0
- **length** (*int*) – field length ??? see dxf-documentation
- **height** (*float*) – textheight in drawing units (default=1)
- **rotation** (*float*) – text rotation (default=0) (all DXF angles in degrees)
- **oblique** (*float*) – text oblique angle in degree, default=0
- **xscale** (*float*) – width factor (default=1)
- **style** (*str*) – textstyle (default=STANDARD)
- **mirror** (*int*) – bit coded flags
- **halign** (*int*) – horizontal justification type, LEFT, CENTER, RIGHT, ALIGN, FIT, BASELINE_MIDDLE (default LEFT)
- **valign** (*int*) – vertical justification type, TOP, MIDDLE, BOTTOM, BASELINE (default BASELINE)
- **alignpoint** – align point (xy- or xyz-tuple), z-axis is 0 by default, if the justification is anything other than BASELINE/LEFT, alignpoint specify the alignment point (or the second alignment point for ALIGN or FIT).

See also:

ATTDEF

DXFEngine.**attrib**(*text*, *insert=(0., 0.)*, ***kwargs*)
Create a new attribute, used in the entities section.

Parameters

- **text** (*str*) – attribute text

- **insert** – insert point (xy- or xyz-tuple), z-axis is 0 by default
- **tag** (*str*) – attribute tag string
- **flags** (*int*) – attribute flags, bit-coded, default=0
- **length** (*int*) – field length ??? see dxf-documentation
- **height** (*float*) – textheight in drawing units (default=1)
- **rotation** (*float*) – text rotation (default=0) (all DXF angles in degrees)
- **oblique** (*float*) – text oblique angle in degree, default=0
- **xscale** (*float*) – width factor (default=1)
- **style** (*str*) – textstyle (default=STANDARD)
- **mirror** (*int*) – bit coded flags
- **halign** (*int*) – horizontal justification type, LEFT, CENTER, RIGHT, ALIGN, FIT, BASELINE_MIDDLE (default LEFT)
- **valign** (*int*) – vertical justification type, TOP, MIDDLE, BOTTOM, BASELINE (default BASELINE)
- **alignpoint** – align point (xy- or xyz-tuple), z-axis is 0 by default, if the justification is anything other than BASELINE/LEFT, alignpoint specify the alignment point (or the second alignment point for ALIGN or FIT).

See also:

ATTRIB

DXFEngine.**block**(*name*, *basepoint=(0., 0.)*, ***kwargs*)

Create a block definition, for the blocks section.

Parameters

- **name** (*str*) – blockname
- **basepoint** – block base point (xy- or xyz-tuple), z-axis is 0. by default
- **flags** (*int*) – block type flags
- **xref** (*str*) – xref pathname

See also:

BLOCK

DXFEngine.**circle**(*radius=1.0*, *center=(0., 0.)*, ***kwargs*)

Create a new circle-entity.

Parameters

- **radius** (*float*) – circle radius
- **center** – center point (xy- or xyz-tuple), z-axis is 0 by default

See also:

CIRCLE

DXFEngine.**face3d**(*points=[]*, ***kwargs*)

Create a 3Dface entity with 3 or 4 sides of (3D) points, z-axis is 0 by default.

Parameters

- **points** – list of three or four 2D- or 3D-points
- **flags** (*int*) – edge flags, bit-coded, default=0

See also:

FACE3D (3DFACE)

DXFEngine.**insert**(*blockname*, *insert=(0., 0.)*, ***kwargs*)
Insert a new block-reference.

Parameters

- **blockname** (*str*) – name of block definition
- **insert** – insert point (xy- or xyz-tuple), z-axis is 0 by default
- **xscale** (*float*) – x-scale factor, default=1.
- **yscale** (*float*) – y-scale factor, default=1.
- **zscale** (*float*) – z-scale factor, default=1.
- **rotation** (*float*) – rotation angle in degree, default=0.
- **columns** (*int*) – column count, default=1
- **rows** (*int*) – row count, default=1
- **colspacing** (*float*) – column spacing, default=0.
- **rowspacing** (*float*) – row spacing, default=0.

DXFEngine.**line**(*start=(0., 0.)*, *end=(0., 0.)*, ***kwargs*)
Create a new line-entity of two (3D) points, z-axis is 0 by default.

Parameters

- **start** – start point (xy- or xyz-tuple)
- **end** – end point (xy- or xyz-tuple)

See also:

LINE

DXFEngine.**point**(*point=(0., 0.)*, ***kwargs*)
Create a new point-entity of one (3D) point, z-axis is 0 by default.

Parameters

- **point** – start point (xy- or xyz-tuple)
- **orientation** – a 3D vector (xyz-tuple), orientation of PDMODE images ... see dxf documentation

See also:

POINT

DXFEngine.**polyline**(*points=[]*, ***kwargs*)
Create a new polyline entity. Polymesh and polyface are also polylines.

Parameters

- **points** – list of points, 2D or 3D points, z-value of 2D points is 0.
- **polyline_elevation** – polyline elevation (xyz-tuple), z-axis supplies elevation, x- and y-axis has to be 0.)

- **flags** (*int*) – polyline flags, bit-coded, default=0
- **startwidth** (*float*) – default starting width, default=0
- **endwidth** (*float*) – default ending width, default=0
- **mcount** (*int*) – polygon mesh M vertex count, default=0
- **ncount** (*int*) – polygon mesh N vertex count, default=0
- **msmooth_density** (*int*) – (if flags-bit POLYLINE_3D_POLYMESH is set) smooth surface M density, default=0
- **nsmooth_density** (*int*) – (if flags-bit POLYLINE_3D_POLYMESH is set) smooth surface N density, default=0 same values as msmooth_density
- **smooth_surface** (*int*) – curves and smooth surface type, default=0 ??? see dxf-documentation

See also:

POLYLINE

DXFEngine.**polymesh**(*nrows*, *ncols*, ***kwargs*)

Create a new polymesh entity.

nrows and ncols >=2 and <= 256, greater meshes have to be divided into smaller meshes.

The flags-bit *POLYLINE_3D_POLYMESH* is set.

Parameters

- **nrows** (*int*) – count of vertices in m-direction, nrows >=2 and <= 256
- **ncols** (*int*) – count of vertices in n-direction, ncols >=2 and <= 256

See also:

POLYMESH

DXFEngine.**polyface**(*precision=6*, ***kwargs*)

Create a new polyface entity, polyface is a dxf-polyline entity!

Parameters **precision** – vertex-coords will be rounded to precision places, and if the vertex is equal to an other vertex, only one vertex will be used, this reduces filespace, the coords will be rounded only for the comparison of the vertices, the output file has the full float resolution.

See also:

POLYFACE

DXFEngine.**shape**(*name*, *insert=(0., 0.)*, ***kwargs*)

Insert a shape-reference.

Parameters

- **name** (*str*) – name of shape
- **insert** – insert point (xy- or xyz-tuple), z-axis is 0 by default
- **xscale** (*float*) – x-scale factor, default=1.
- **rotation** (*float*) – rotation angle in degree, default=0
- **oblique** (*float*) – text oblique angle in degree, default=0

See also:

SHAPE

DXFEngine.**solid**(*points=[]*, ***kwargs*)
Create a solid-entity by 3 or 4 vertices, the z-axis for 2D-points is 0.

Parameters **points** (*list*) – three or four 2D- or 3D-points (tuples)

See also:

SOLID

DXFEngine.**trace**(*points=[]*, ***kwargs*)
Create a trace-entity by 3 or 4 vertices, the z-axis for 2D-points is 0.

Parameters **points** – list of three or four 2D- or 3D-points (tuples)

See also:

TRACE

DXFEngine.**text**(*text*, *insert=(0.*, *0.)*, *height=1.0*, ***kwargs*)
Create a new text entity.

Parameters

- **text** (*str*) – the text to display
- **insert** – insert point (xy- or xyz-tuple), z-axis is 0 by default
- **height** (*float*) – text height in drawing-units
- **rotation** (*float*) – text rotation in degree, default=0
- **xscale** (*float*) – text width factor, default=1
- **oblique** (*float*) – text oblique angle in degree, default=0
- **style** (*str*) – text style name, default=STANDARD
- **mirror** (*int*) – text generation flags, bit-coded, default=0
- **halign** (*int*) – horizontal justification type
- **valign** (*int*) – vertical justification type
- **alignpoint** – align point (xy- or xyz-tuple), z-axis is 0 by default If the justification is anything other than BASELINE/LEFT, alignpoint specify the alignment point (or the second alignment point for ALIGN or FIT).

any combination of *valign* (TOP, MIDDLE, BOTTOM) and *halign* (LEFT, CENTER, RIGHT) is valid.

See also:

TEXT

DXFEngine.**viewport**(*center_point*, *width*, *height*, ***kwargs*)
Create a new viewport entity.

Parameters

- **center_point** – center point of viewport in paper space as (x, y, z) tuple
- **width** (*float*) – width of viewport in paper space
- **height** (*float*) – height of viewport in paper space
- **status** (*int*) – 0 for viewport is off, >0 'stacking' order, 1 is highest priority
- **view_target_point** – as (x, y, z) tuple, default value is (0, 0, 0)
- **view_direction_vector** – as (x, y, z) tuple, default value is (0, 0, 0)

- **view_twist_angle** (*float*) – in degrees, default value is 0
- **view_height** (*float*) – default value is 1
- **view_center_point** – as (x, y) tuple, default value is (0, 0)
- **perspective_lens_length** (*float*) – default value is 50
- **front_clip_plane_z_value** (*float*) – default value is 0
- **back_clip_plane_z_value** (*float*) – default value is 0
- **view_mode** (*int*) – default value is 0
- **circle_zoom** (*int*) – default value is 100
- **fast_zoom** (*int*) – default value is 1
- **ucs_icon** (*int*) – default value is 3
- **snap** (*int*) – default value is 0
- **grid** (*int*) – default value is 0
- **snap_style** (*int*) – default value is 0
- **snap_isopair** (*int*) – default value is 0
- **snap_angle** (*float*) – in degrees, default value is 0
- **snap_base_point** – as (x, y) tuple, default value is (0, 0)
- **snap_spacing** – as (x, y) tuple, default value is (0.1, 0.1)
- **grid_spacing** – as (x, y) tuple, default value is (0.1, 0.1)
- **hidden_plot** (*int*) – default value is 0

See also:

VIEWPORT (Entity)

Composite Entities

DXFEngine.**mtext**(*text*, *insert*, *linespacing=1.5*, ***kwargs*)

Create a multi-line text buildup *MText* with simple *TEXT* entities.

Mostly the same kwargs like *TEXT*.

> **Caution:** **alignpoint** is always the insert point, I don't need a second alignpoint because horizontal alignment FIT, ALIGN, BASELINE_MIDDLE is not supported.

Parameters

- **text** (*str*) – the text to display
- **insert** – insert point (xy- or xyz-tuple), z-axis is 0 by default
- **linespacing** (*float*) – linespacing in percent of height, 1.5 = 150% = 1+1/2 lines
- **height** (*float*) – text height in drawing-units
- **rotation** (*float*) – text rotion in dregree, default=0
- **xscale** (*float*) – text width factor, default=1

- **oblique** (*float*) – text oblique angle in degree, default=0
- **style** (*str*) – text style name, default=STANDARD
- **mirror** (*int*) – text generation flags, bit-coded, default=0
- **halign** (*int*) – horizontal justification type
- **valign** (*int*) – vertical justification type
- **layer** (*str*) – layer name
- **color** (*int*) – range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*

any combination of *valign* (TOP, MIDDLE, BOTTOM) and *halign* (LEFT, CENTER, RIGHT) is valid.

See also:

MText

DXFEngine.**insert2**(*blockdef*, *insert=(0., 0.)*, *attribs={}*, ***kwargs*)

Insert a new block-reference with auto-creating of *ATTRIB* from *ATTDEF*, and setting attrib-text by the attribs-dict. (multi-insert is not supported)

Parameters

- **blockdef** – the block definition itself
- **insert** – insert point (xy- or xyz-tuple), z-axis is 0 by default
- **xscale** (*float*) – x-scale factor, default=1.
- **yscale** (*float*) – y-scale factor, default=1.
- **zscale** (*float*) – z-scale factor, default=1.
- **rotation** (*float*) – rotation angle in degree, default=0.
- **attribs** (*dict*) – dict with tag:value pairs, to fill the the attdefs in the block-definition. example: {'TAG1': 'TextOfTAG1'}, create and insert an attrib from an attdef (with tag-value == 'TAG1'), and set text-value of the attrib to value 'TextOfTAG1'.
- **linetype** (*str*) – linetype name, if not defined = *BYLAYER*
- **layer** (*str*) – layer name
- **color** (*int*) – range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*

See also:

Insert2

DXFEngine.**table**(*insert*, *nrows*, *ncols*, *default_grid=True*)

Table object like a HTML-Table, buildup with basic DXF R12 entities.

Cells can contain Multiline-Text or DXF-BLOCKs, or you can create your own cell-type by extending the CustomCell object.

Cells can span over columns and rows.

Text cells can contain text with an arbitrary rotation angle, or letters can be stacked top-to-bottom.

BlockCells contains block references (INSERT-entity) created from a block definition (BLOCK), if the block definition contains attribute definitions (ATTDEF-entity), attribs created by Attdef.new_attrib() will be added to the block reference (ATTRIB-entity).

Parameters

- **insert** – insert point as 2D or 3D point
- **nrows** (*int*) – row count
- **ncols** (*int*) – column count
- **default_grid** (*bool*) – if *True* always a solid line grid will be drawn, if *False*, only explicit defined borders will be drawn, default grid has a priority of 50.

See also:

Table

DXFEngine.**rectangle**(*insert*, *width*, *height*, ***kwargs*)

2D Rectangle, build with a polyline and a solid as background filling

Parameters

- **insert** (point) – where to place the rectangle
- **width** (*float*) – width in drawing units
- **height** (*float*) – height in drawing units
- **rotation** (*float*) – in degree (circle = 360 degree)
- **halign** (*int*) – *LEFT*, *CENTER*, *RIGHT*
- **valign** (*int*) – *TOP*, *MIDDLE*, *BOTTOM*
- **color** (*int*) – dxf color index, default is *BYLAYER*, if color is None, no polyline will be created, and the rectangle consist only of the background filling (if bgcolor != *None*)
- **bgcolor** (*int*) – dxf color index, default is *None* (no background filling)
- **layer** (*str*) – target layer, default is '0'
- **linetype** (*str*) – linetype name, None = *BYLAYER*

See also:

Rectangle

DXFEngine.**ellipse**(*center*, *rx*, *ry*, *startangle=0.*, *endangle=360.*, *rotation=0.*, *segments=100*, ***kwargs*)

Create a new ellipse-entity, curve shape is an approximation by *POLYLINE*.

Parameters

- **center** – center point (xy- or xyz-tuple), z-axis is 0 by default
- **rx** (*float*) – radius in x-axis
- **ry** (*float*) – radius in y-axis
- **startangle** (*float*) – in degree
- **endangle** (*float*) – in degree
- **rotation** (*float*) – angle between x-axis and ellipse-main-axis in degree
- **segments** (*int*) – count of line segments for polyline approximation
- **linetype** (*str*) – linetype name, if not defined = *BYLAYER*
- **layer** (*str*) – layer name
- **color** (*int*) – range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*

See also:

Ellipse

DXFEngine.**spline**(*points*, *segments=100*, ***kwargs*)

Create a new cubic-spline-entity, curve shape is an approximation by *POLYLINE*.

Parameters

- **points** – breakpoints (knots) as 2D points (float-tuples), defines the curve, the curve goes through this points
- **segments** (*int*) – count of line segments for polyline approximation
- **linetype** (*str*) – linetype name, if not defined = *BYLAYER*
- **layer** (*str*) – layer name
- **color** (*int*) – range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*

See also:

Spline

DXFEngine.**bezier**(***kwargs*)

Create a new cubic-bezier-entity, curve shape is an approximation by *POLYLINE*.

Parameters

- **linetype** (*str*) – linetype name, if not defined = *BYLAYER*
- **layer** (*str*) – layer name
- **color** (*int*) – range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*

See also:

Bezier

DXFEngine.**clothoid**(*start=(0, 0)*, *rotation=0.*, *length=1.*, *paramA=1.0*, *mirror=""*, *segments=100*, ***kwargs*)

Create a new clothoid-entity, curve shape is an approximation by *POLYLINE*.

Parameters

- **start** – insert point as 2D points (float-tuples)
- **rotation** (*float*) – in degrees
- **length** (*loat*) – length of curve in drawing units
- **paramA** (*float*) – clothoid parameter A
- **mirror** (*str*) – `'x'` for mirror curve about x-axis, `'y'` for mirror curve about y-axis, `'xy'` for mirror curve about x- and y-axis
- **segments** (*int*) – count of line segments for polyline approximation
- **linetype** (*str*) – linetype name, if not defined = *BYLAYER*
- **layer** (*str*) – layer name
- **color** (*int*) – range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*

See also:

Clothoid

HEADER Section

The HEADER section of the DXF file contains settings of variables associated with the drawing. The following list shows the header variables and their meanings.

set/get header variables:

```
#set value
drawing.header['$ANGBASE'] = 30
drawing.header['$EXTMIN'] = (0, 0, 0)
drawing.header['$EXTMAX'] = (100, 100, 0)

#get value
version = drawing.header['$ACADVER'].value

# for 2D/3D points use:
minx, miny, minz = drawing.header['$EXTMIN'].tuple
maxx, maxy, maxz = drawing.header['$EXTMAX'].tuple
```

Variable	Type	Description
$ACADVER	string	The AutoCAD drawing database version number; AC1006 = R10, AC1009 = R11 and R12
$ANGBASE	float	Angle 0 direction
$ANGDIR	int	1 = clockwise angles, 0 = counterclockwise
$ATTDIA	int	Attribute entry dialogs, 1 = on, 0 = off
$ATTMODE	int	Attribute visibility: 0 = none, 1 = normal, 2 = all
$ATTREQ	int	Attribute prompting during INSERT, 1 = on, 0 = off
$AUNITS	int	Units format for angles
$AUPREC	int	Units precision for angles
$AXISMODE	int	Axis on if nonzero (not functional in Release 12)
$AXISUNIT	2DPoint	Axis X and Y tick spacing (not functional in Release 12)
$BLIPMODE	int	Blip mode on if nonzero
$CECOLOR	int	Entity color number; 0 = BYBLOCK, 256 = BYLAYER
$CELTYPE	string	Entity linetype name, or BYBLOCK or BYLAYER
$CHAMFERA	float	First chamfer distance
$CHAMFERB	float	Second chamfer distance
$CLAYER	string	Current layer name
$COORDS	int	0 = static coordinate display, 1 = continuous update, 2 = “d<a” format
$DIMALT	int	Alternate unit dimensioning performed if nonzero
$DIMALTD	int	Alternate unit decimal places
$DIMALTF	float	Alternate unit scale factor
$DIMAPOST	string	Alternate dimensioning suffix
$DIMASO	int	1 = create associative dimensioning, 0 = draw individual entities
$DIMASZ	float	Dimensioning arrow size
$DIMBLK	string	Arrow block name
$DIMBLK1	string	First arrow block name
$DIMBLK2	string	Second arrow block name
$DIMCEN	float	Size of center mark/lines
$DIMCLRD	int	Dimension line color, range is 0 = BYBLOCK, 256 = BYLAYER
$DIMCLRE	int	Dimension extension line color, range is 0 = BYBLOCK, 256 = BYLAYER
$DIMCLRT	int	Dimension text color, range is 0 = BYBLOCK, 256 = BYLAYER
$DIMDLE	float	Dimension line extension
$DIMDLI	float	Dimension line increment

Table 1.1 – continued from

Variable	Type	Description
$DIMEXE	float	Extension line extension
$DIMEXO	float	Extension line offset
$DIMGAP	float	Dimension line gap
$DIMLFAC	float	Linear measurements scale factor
$DIMLIM	int	Dimension limits generated if nonzero
$DIMPOST	string	General dimensioning suffix
$DIMRND	float	Rounding value for dimension distances
$DIMSAH	int	Use separate arrow blocks if nonzero
$DIMSCALE	float	Overall dimensioning scale factor
$DIMSE1	int	First extension line suppressed if nonzero
$DIMSE2	int	Second extension line suppressed if nonzero
$DIMSHO	int	1 = Recompute dimensions while dragging, 0 = drag original image
$DIMSOXD	int	Suppress outside-extensions dimension lines if nonzero
$DIMSTYLE	string	Dimension style name
$DIMTAD	int	Text above dimension line if nonzero
$DIMTFAC	float	Dimension tolerance display scale factor
$DIMTIH	int	Text inside horizontal if nonzero
$DIMTIX	int	Force text inside extensions if nonzero
$DIMTM	float	Minus tolerance
$DIMTOFL	int	If text outside extensions, force line extensions between extensions if nonzero
$DIMTOH	int	Text outside horizontal if nonzero
$DIMTOL	int	Dimension tolerances generated if nonzero
$DIMTP	float	Plus tolerance
$DIMTSZ	float	Dimensioning tick size: 0 = no ticks
$DIMTVP	float	Text vertical position
$DIMTXT	float	Dimensioning text height
$DIMZIN	int	Zero suppression for “feet & inch” dimensions
$DWGCODEPAGE	int	Drawing code page. Set to the system code page when a new drawing is created, but not otherwise
$DRAGMODE	int	0 = off, 1 = on, 2 = auto
$ELEVATION	float	Current elevation set by ELEV command
$EXTMAX	3DPoint	X, Y, and Z drawing extents upper-right corner (in WCS)
$EXTMIN	3DPoint	X, Y, and Z drawing extents lower-left corner (in WCS)
$FILLETRAD	float	Fillet radius
$FILLMODE	int	Fill mode on if nonzero
$HANDLING	int	Handles enabled if nonzero
$HANDSEED	string	Next available handle
$INSBASE	3DPoint	Insertion base set by BASE command (in WCS)
$LIMCHECK	int	Nonzero if limits checking is on
$LIMMAX	2DPoint	XY drawing limits upper-right corner (in WCS)
$LIMMIN	2DPoint	XY drawing limits lower-left corner (in WCS)
$LTSCALE	float	Global linetype scale
$LUNITS	int	Units format for coordinates and distances
$LUPREC	int	Units precision for coordinates and distances
$MAXACTVP	int	Sets maximum number of viewports to be regenerated
$MENU	string	Name of menu file
$MIRRTEXT	int	Mirror text if nonzero
$ORTHOMODE	int	Ortho mode on if nonzero
$OSMODE	int	Running object snap modes
$PDMODE	int	Point display mode

Table 1.1 – continued from

Variable	Type	Description
$PDSIZE	float	Point display size
$PELEVATION	float	Current paper space elevation
$PEXTMAX	3DPoint	Maximum X, Y, and Z extents for paper space
$PEXTMIN	3DPoint	Minimum X, Y, and Z extents for paper space
$PLIMCHECK	int	Limits checking in paper space when nonzero
$PLIMMAX	2DPoint	Maximum X and Y limits in paper space
$PLIMMIN	2DPoint	Minimum X and Y limits in paper space
$PLINEGEN	int	Governs the generation of linetype patterns around the vertices of a 2D Polyline 1 = linetype is gen
$PLINEWID	float	Default Polyline width
$PSLTSCALE	int	Controls paper space linetype scaling 1 = no special linetype scaling 0 = viewport scaling governs l
$PUCSNAME	string	Current paper space UCS name
$PUCSORG	3DPoint	Current paper space UCS origin
$PUCSXDIR	3DPoint	Current paper space UCS X axis
$PUCSYDIR	3DPoint	Current paper space UCS Y axis
$QTEXTMODE	int	Quick text mode on if nonzero
$REGENMODE	int	REGENAUTO mode on if nonzero
$SHADEDGE	int	0 = faces shaded, edges not highlighted 1 = faces shaded, edges highlighted in black 2 = faces not f
$SHADEDIF	int	Percent ambient/diffuse light, range 1-100, default 70
$SKETCHINC	float	Sketch record increment
$SKPOLY	int	0 = sketch lines, 1 = sketch polylines
$SPLFRAME	int	Spline control polygon display, 1 = on, 0 = off
$SPLINESEGS	int	Number of line segments per spline patch
$SPLINETYPE	int	Spline curve type for PEDIT Spline (See your AutoCAD Reference Manual)
$SURFTAB1	int	Number of mesh tabulations in first direction
$SURFTAB2	int	Number of mesh tabulations in second direction
$SURFTYPE	int	Surface type for PEDIT Smooth (See your AutoCAD Reference Manual)
$SURFU	int	Surface density (for PEDIT Smooth) in M direction
$SURFV	int	Surface density (for PEDIT Smooth) in N direction
$TDCREATE	float	Date/time of drawing creation
$TDINDWG	float	Cumulative editing time for this drawing
$TDUPDATE	float	Date/time of last drawing update
$TDUSRTIMER	float	User elapsed timer
$TEXTSIZE	float	Default text height
$TEXTSTYLE	string	Current text style name
$THICKNESS	float	Current thickness set by ELEV command
$TILEMODE	int	1 for previous release compatibility mode, 0 otherwise
$TRACEWID	float	Default Trace width
$UCSNAME	string	Name of current UCS
$UCSORG	3DPoint	Origin of current UCS (in WCS)
$UCSXDIR	3DPoint	Direction of current UCS's X axis (in World coordinates)
$UCSYDIR	3DPoint	Direction of current UCS's Y axis (in World coordinates)
$UNITMODE	int	Low bit set = display fractions, feet-and-inches, and surveyor's angles in input format
$USERI1 - 5	int	Five integer variables intended for use by third-party developers
$USERR1 - 5	float	Five real variables intended for use by third-party developers
$USRTIMER	int	0 = timer off, 1 = timer on
$VISRETAIN	int	0 = don't retain Xref-dependent visibility settings, 1 = retain Xref-dependent visibility settings
$WORLDVIEW	int	1 = set UCS to WCS during DVIEW/VPOINT, 0 = don't change UCS

DXFTypes

In normal cases you **don't** get in touch with *DXFTypes*.

DXFList

DXFList can contain every dxf drawing entity. In the usual case DXFList is used to group lines, arcs, circles and similar entities, also all composite entities (Table, MText, Rectangle) can be used. Add with **append(entity)** single entities to the list.

usage:

```
from dxfwrite import DXFEngine as dxf

drawing = dxf.drawing()
entities = DXFList()
entities.append(dxf.line((0, 0), (10, 0)))
entities.append(dxf.text('Text'))
drawing.add(entities)
drawing.saveas('test.dxf')
```

DXFString

Create a basic dxf string with the default group code 1.

usage:

```
string = DXFString(value='VARNAME', group_code=1)
```

DXFName

Create a basic dxf string with the default group code 2.

DXFFloat

Create a basic dxf float with the default group code 40.

DXFAngle

Create a basic dxf float with the default group code 50.

DXFInt

Create a basic dxf integer with the default group code 70.

DXFBool

Create a basic dxf bool (0 or 1 as integer) with the default group code 290.

DXFPoint

Create a basic dxf point., a dxf point consist of three floats with the following group codes:

- x-coordinate – 10+index_shift
- y-coordinate – 20+index_shift
- z-coordinate – 30+index_shift

Access coordinates by the index operator:

```
point = DXFPoint(coords=(x, y, z), index_shift=0)
x = point['x'] # or point[0]
y = point['y'] # or point[1]
z = point['z'] # or point[2]
x, y = point['xy']
x, y, z = point['xyz']
z, y, x = point['zyx']
```

DXFPoint2D

like DXFPoint, but only x and y coordinates will be used.

DXFPoint3D

like DXFPoint, but alway 3 coordinates will be used, z = 0 if omitted.

LAYER

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.

Create a layer:

```
drawing.add_layer(name)
```

is a shortcut for:

```
layer = DXFEngine.layer(name)
drawing.layers.add(layer)
```

DXFEngine.**layer**(*name*, ***kwargs*)

Parameters

- **name** (*string*) – layer name
- **flags** (*int*) – standard flag values, bit-coded, default=0
- **color** (*int*) – color number, negative if layer is off, default=1
- **linetype** (*string*) – linetype name, default="CONTINUOUS"

Flags

Flag	Description
LAYER_FROZEN	If set, layer is frozen
LAYER_FROZEN_BY_DEFAULT	If set, layer is frozen by default in new Viewports
LAYER_LOCKED	If set, layer is locked

Get *Layer* `'name'` from Drawing:

```
layer = drawing.layers['name']
layer.off()
```

class Layer

Defines the layer properties. Get/set properties by the subscript operator *[ ]*.

- `layer['name']`
- `layer['flags']` (for status change see also special methods below)
- `layer['color']`
- `layer['linetype']`

`Layer.`**`on`**`()`

`Layer.`**`off`**`()`

`Layer.`**`freeze`**`()`

`Layer.`**`thaw`**`()`

`Layer.`**`lock`**`()`

`Layer.`**`unlock`**`()`

Example:

```
from dxfwrite import DXFEngine as dxf

drawing = dxf.drawing('drawing.dxf')
drawing.add_layer('LINES', color=3, linetype='DASHED')
line = dxf.line((1.2, 3.7), (5.5, 9.7), layer='LINES')
drawing.add(line)
drawing.save()
```

TEXTSTYLE

The DXF format assigns text properties to individual lines of text based on text styles. These text styles are similar to the paragraph styles in a word processor; they contain font and other settings that determine the look and feel of text.

A DXF text style includes:

- The font
- A text height, which you can set or leave at 0 for later flexibility
- Special effects (where available), such as italic
- Really special effects, such as vertical and upside down

To use the textstyles just assign the stylename as string.

Predefined text styles for all drawings created with dxfwrite:

Stylename	True Type Font
STANDARD	arial.ttf
ARIAL	arial.ttf
ARIAL_BOLD	arialbd.ttf
ARIAL_ITALIC	ariali.ttf
ARIAL_BOLD_ITALIC	arialbi.ttf
ARIAL_BLACK	ariblk.ttf
ISOCPEUR	isocpeur.ttf
ISOCPEUR_ITALIC	isocpeui.ttf
TIMES	times.ttf
TIMES_BOLD	timesbd.ttf
TIMES_ITALIC	timesi.ttf
TIMES_BOLD_ITALIC	timesbi.ttf

Create a Textstyle:

```
drawing.add_style(stylename)
```

is a shortcut for:

```
style = DXFEngine.style(name)
drawing.styles.add(style)
```

DXFEngine.**style**(*name*, ***kwargs*)

Parameters

- **name** (*string*) – textstyle name
- **flags** (*int*) – standard flag values, bit-coded, default=0
- **generation_flags** (*int*) – text generation flags, default = 0
- **height** (*float*) – fixed text height, 0 if not fixed = default
- **last_height** – last height used, default=1.
- **width** (*float*) – width factor, default=1.
- **oblique** (*float*) – oblique angle in degree, default=0.
- **font** (*string*) – primary font filename, default="ARIAL"
- **bigfont** (*string*) – big-font file name, default=""

Generation Flags

Flag	Description
STYLE_TEXT_BACKWARD	Text is backward (mirrored in X)
STYLE_TEXT_UPSIDEDOWN	Text is upside down (mirrored in Y)

A text style **height of 0.0** makes the style variable height, which means that you can specify the height separately for each text object. Assigning a fixed (that is, nonzero) height to a text style forces all text using the style to be the same height. Variable height styles are more flexible, but fixed height styles usually make it easier to draw text of consistent height.

LINETYPE

A Linetype defines a line pattern, which can be used by DXF entities. A Linepattern can contain solid line elements, points and gaps (see *LINEPATTERN*).

Create a linetype:

```
drawing.add_linetype(name, pattern=linepattern)
```

is a shortcut for:

```
linetype = DXFEngine.linetype(name, pattern=linepattern)
drawing.linetypes.add(linetype)
```

DXFEngine.**linetype**(*name*, ***kwargs*)

Parameters

- **name** (*string*) – linetype name
- **flags** (*int*) – standard flag values, bit-coded, default=0
- **description** (*string*) – descriptive text for linetype, default=""
- **pattern** – line pattern definition, see method *DXFEngine.linepattern*

The following Linetypes are predefined by dxfwrite:

```
CONTINUOUS: Solid
CENTER: Center ____ _ ____ _ ____ _ ____ _ ____ _ ____
CENTERX2: Center (2x) ________  __  ________  __  ________
CENTER2: Center (.5x) ____ _ ____ _ ____ _ ____ _ ____
DASHED: Dashed __ __ __ __ __ __ __ __ __ __ __ __ _
DASHEDX2: Dashed (2x) ____  ____  ____  ____  ____  ____
DASHED2:Dashed (.5x) _ _ _ _ _ _ _ _ _ _ _ _ _ _
PHANTOM: Phantom ______  __  __  ______  __  __  ______
PHANTOMX2: Phantom (2x)______________  ____  ____  ______________
PHANTOM2: Phantom (.5x) ___ _ _ ___ _ _ ___ _ _ ___ _ _ ___
DASHDOT: Dash dot __ . __ . __ . __ . __ . __ . __ . __
DASHDOTX2: Dash dot (2x) ____  .  ____  .  ____  .  ____
DASHDOT2: Dash dot (.5x) _ . _ . _ . _ . _ . _ . _ . _
DOT: Dot .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
DOTX2: Dot (2x) .  .  .  .  .  .  .  .  .
DOT2: Dot (.5) . . . . . . . . . . . . . . . . . . . .
DIVIDE: Divide __ . . __ . . __ . . __ . . __ . . __
DIVIDEX2: Divide (2x) ____  . .  ____  . .  ____  . .  ____
DIVIDE2: Divide (.5) _ . _ . _ . _ . _ . _ . _ . _
```

LINEPATTERN

Create a linepattern:

```
linepattern = DXFEngine.linepattern(pattern)
```

DXFEngine.**linepattern**(*pattern*)

Parameters **pattern** – is a list of float values, elements > 0 are solid line segments, elements < 0 are gaps and elements = 0 are points. pattern[0] = total pattern length in drawing units

example linepattern([2.0, 1.25, -0.25, 0, -0.25])

VIEW

A view is a named 'look' at the drawing model. When you create a specific views by name, you can use them for layout or when you need to refer to specific details. A named view consists of a specific magnification, position, and orientation.

Create a view:

```
drawing.add_view(name, ...)
```

is a shortcut for:

```
view = DXFEngine.view(name, ...)
drawing.views.add(view)
```

DXFEngine.**view**(*name*, ***kwargs*)

Parameters

- **name** (*string*) – view name
- **flags** (*int*) – standard flag values, bit-coded, default=0 STD_FLAGS_PAPER_SPACE, if set this is a paper space view.
- **height, width** (*float*) – view height and width, in DCS?!, default=1.0
- **center_point** – view center point, in DCS?! (xy-tuple), default=(.5, .5)
- **direction_point** – view direction from target point, in WCS!! (xyz-tuple), default=(0, 0, 1)
- **target_point** – target point, in WCS!! (xyz-tuple), default=(0, 0, 0)
- **lens_length** (*float*) – lens length, default=50
- **front_clipping** (*float*) – front and back clipping planes, offsets from target point, default=0
- **back_clipping** – see front_clipping
- **view_twist** (*float*) – twist angle in degree, default=0
- **view_mode** (*int*) – view mode, bit-coded, default=0

View Mode Flags

Flags	Description
VMODE_TURNED_OFF	view is turned off if bit is set
VMODE_PERSPECTIVE_VIEW_ACTIVE	view is in perspective mode if bit is set
VMODE_FRONT_CLIPPING_ON	front clipping is on if bit is set
VMODE_BACK_CLIPPING_ON	back clipping is on if bit ist set
VMODE_UCS_FOLLOW_MODE_ON	???
VMODE_FRONT_CLIP_NOT_AT_EYE	???

VIEWPORT (Table Entry)

A viewport is a windows containing a view to the drawing model. You can change the default view, which will be displayed on opening the drawing with a CAD program, by adding a viewport named `'*ACTIVE'`. In AutoCAD you can place multiple viewports in the main editor window (Left, Right, Top), but don't ask me how to do this in a DXF file.

Create a viewport:

```
drawing.add_vport(name, ...)
```

is a shortcut for:

```
vport = DXFEngine.vport(name, ...)
drawing.viewports.add(vport)
```

DXFEngine.**vport**(*name*, ***kwargs*)

Parameters

- **name** (*str*) – viewport name
- **flags** (*int*) – standard flag values, bit-coded, default=0
- **lower_left** – lower-left corner of viewport, (xy-tuple), default=(0, 0)
- **upper_right** – upper-right corner of viewport, (xy-tuple), default=(1, 1)
- **center_point** – view center point, in WCS, (xy-tuple), default=(.5, .5)
- **snap_base** – snap base point, (xy-tuple), default=(0, 0)
- **snap_spacing** – snap spacing, X and Y (xy-tuple), default=(.1, .1)
- **grid_spacing** – grid spacing, X and Y (xy-tuple), default=(.1, .1)
- **direction_point** – view direction from target point (xyz-tuple), default=(0, 0, 1)
- **target_point** – view target point (xyz-tuple), default=(0, 0, 0)
- **aspect_ratio** – viewport aspect ratio (float), default=1.
- **lens_length** (*float*) – lens length, default=50
- **front_clipping** (*float*) – front and back clipping planes, offsets from target point , default=0
- **back_clipping** (*float*) – see front_clipping
- **view_twist** (*float*) – twist angle in degree, default=0
- **circle_zoom** (*float*) – circle zoom percent, default=100

- **view_mode** (*int*) – view mode, bit-coded, default=0
- **fast_zoom** (*int*) – fast zoom setting, default=1
- **ucs_icon** (*int*) – UCSICON settings, default=3
- **snap_on** (*int*) – snap on/off, default=0
- **grid_on** (*int*) – grid on/off, default=0
- **snap_style** (*int*) – snap style, default=0
- **snap_isopair** (*int*) – snap isopair, default=0

View Mode Flags

Flags	Description
VMODE_TURNED_OFF	viewport is turned off if bit is set
VMODE_PERSPECTIVE_VIEW_ACTIVE	viewport is in perspective mode if bit is set
VMODE_FRONT_CLIPPING_ON	front clipping is on if bit is set
VMODE_BACK_CLIPPING_ON	back clipping is on if bit is set
VMODE_UCS_FOLLOW_MODE_ON	???
VMODE_FRONT_CLIP_NOT_AT_EYE	???

CHAPTER 2

DXF R12 Entities

ARC

Type: Basic DXF R12 entity.

Draws circular arcs — arcs cut from circles, not from ellipses, parabolas, or some other complicated curve, the arc goes from start angle to end angle.

`DXFEngine.`**`arc`**(*radius=1.0*, *center=(0., 0.)*, *startangle=0.*, *endangle=360.*, ***kwargs*)

Parameters

- **`radius`** (*`float`*) – arc radius
- **`center`** – center point (xy- or xyz-tuple), z-axis is 0 by default
- **`startangle`** (*`float`*) – start angle in degree
- **`endangle`** (*`float`*) – end angle in degree

Common Keyword Arguments for all Basic DXF R12 Entities

keyword	description
layer	Layer name as string
linetype	Linetype name as string, if not defined = *BYLAYER*
color	as integer in range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*
thickness	Thickness as float
paper_space	0 = entity is in model_space, 1 = entity is in paper_space
extrusion_direction	3D Point as tuple(x, y, z) if extrusion direction is not parallel to the World Z axis

Attribs of DXF entities can be changed by the index operator:

```
from dxfwrite import DXFEngine as dxf

drawing = dxf.drawing('drawing.dxf')
```

```
arc = dxf.arc(2.0, (1.0, 1.0), 30, 90)
arc['layer'] = 'points'
arc['color'] = 7
arc['center'] = (2, 3, 7) # int or float
arc['radius'] = 3.5
drawing.add(arc)
drawing.save()
```

ATTDEF

Type: Basic DXF R12 entity.

Create a new attribute definition, you can use in the *block definition*.

You create an attribute definition, which acts as a placeholder for a text string that can vary each time you insert the block. You include the attribute definition when you create the *block definition*. Then each time you *insert* the block, you can create an new attribute from the attribute definition and add them to the block-reference.

After you define the attribute definition you can create a new *Attrib* and insert it into a *block reference*, you can just use the `attdef.new_attrib()` method and change all the default values preset from the *ATTDEF* object.

You rarely need to use any of the flags settings (Invisible, Constant, Verify, or Preset).

`DXFEngine.`**`attdef`**(*tag*, *insert=(0., 0.)*, ***kwargs*)

Parameters

- **`text`** (*`str`*) – attribute default text
- **`insert`** – insert point (xy- or xyz-tuple), z-axis is 0 by default
- **`prompt`** (*`str`*) – prompt text, like “insert a value:”
- **`tag`** (*`str`*) – attribute tag string
- **`flags`** (*`int`*) – attribute flags, bit-coded, default=0
- **`length`** (*`int`*) – field length ??? see dxf-documentation
- **`height`** (*`float`*) – textheight in drawing units (default=1)
- **`rotation`** (*`float`*) – text rotation (default=0) (all DXF angles in degrees)
- **`oblique`** (*`float`*) – text oblique angle in degree, default=0
- **`xscale`** (*`float`*) – width factor (default=1)
- **`style`** (*`str`*) – textstyle (default=STANDARD)
- **`mirror`** (*`int`*) – bit coded flags
- **`halign`** (*`int`*) – horizontal justification type, LEFT, CENTER, RIGHT, ALIGN, FIT, BASELINE_MIDDLE (default LEFT)
- **`valign`** (*`int`*) – vertical justification type, TOP, MIDDLE, BOTTOM, BASELINE (default BASELINE)
- **`alignpoint`** – align point (xy- or xyz-tuple), z-axis is 0 by default, if the justification is anything other than BASELINE/LEFT, alignpoint specify the alignment point (or the second alignment point for ALIGN or FIT).

Flags

Flags	Description
ATTRIB_IS_INVISIBLE	Attribute is invisible (does not display)
ATTRIB_IS_CONST	This is a constant Attribute
ATTRIB_REQUIRE_VERIFICATION	Verification is required on input of this Attribute
ATTRIB_IS_PRESET	Verification is required on input of this Attribute

Mirror Flags

Flags	Description
dxfwrite.MIRROR_X	Text is backward (mirrored in X)
dxfwrite.MIRROR_Y	Text is upside down (mirrored in Y)

Common Keyword Arguments for all Basic DXF R12 Entities

keyword	description
layer	Layer name as string
linetype	Linetype name as string, if not defined = *BYLAYER*
color	as integer in range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*
thickness	Thickness as float
paper_space	0 = entity is in model_space, 1 = entity is in paper_space
extrusion_direction	3D Point as tuple(x, y, z) if extrusion direction is not parallel to the World Z axis

Methods

Attdef.**new_attrib**(***kwargs*)

Create a new ATTRIB with attdef's attributs as default values.

Parameters **kwargs** – override the attdef default values.

example:

```
from dxfwrite import DXFEngine as dxf
drawing = dxf.drawing('test.dxf')
block = dxf.block(name='BLOCK1')
attdef = dxf.attdef(insert=(.2, .2),
    rotation=30,
    height=0.25,
    text='test',  # default text
    prompt='input text:', # only important for interactive CAD systems
    tag='BLK')
block.add(attdef)
drawing.block.add(block) # add block definition to drawing
blockref = dxf.insert(blockname='BLOCK1', insert=(10, 10)) # create a block reference
# create a new attribute, given keywords override the default values from the attrib␣
↪definition
attrib = attdef.new_attrib(height=0.18, text='TEST')
# add the attrib to the block reference, insert has the default value (.2, .2),
# and insert is relative to block insert point
blockref.add(attrib, relative=True)
drawing.add(blockref) # add block reference to drawing
drawing.save()
```

ATTRIB

Type: Basic DXF R12 entity.

Create a new attribute, attach this attribute to a block-reference that you made previously.

Attributes are fill-in-the-blank text fields that you can add to your blocks. When you create a block definition and then insert it several times in a drawing, all the ordinary geometry (lines, circles, regular text strings, and so on) in all the instances are exactly identical. Attributes provide a little more flexibility in the form of text strings that can be different in each block insert.

1. First you have to create the *ATTDEF*.
2. Next you will create the block and add the *ATTDEF* with the *block.add(attdef)* method.
3. Create a block-reference *blockref=DXFEngine.insert(blockname, insert)* by *INSERT*.
4. Create an *attrib = attdef.new_attrib(kwargs)*
5. Add *attrib* to block-reference by *blockref.add(attrib)*
6. Add *blockref* to the dxf-drawing, *drawing.add(blockref)*

When you create attributes you can put them on their own layer. This makes it easy to hide them or display them by turning the layer they are on off. This is handy when you are using attributes to hold information like phone numbers on a desk floor plan. Sometimes you will want to see, and plot, the desks without the text.

Probably the most interesting application for attributes is that you can use them to create tables and reports that accurately reflect the information you have stored in your blocks, but this works only in CAD Applications, not with dxfwrite. The process for doing this is somewhat complex and depends on the used CAD-Application.

DXFEngine.**attrib**(*text*, *insert=(0., 0.)*, ***kwargs*)

Create a new attribute, used in the entities section.

Parameters

- **text** (*str*) – attribute text
- **insert** – insert point (xy- or xyz-tuple), z-axis is 0 by default
- **tag** (*str*) – attribute tag string
- **flags** (*int*) – attribute flags, bit-coded, default=0
- **length** (*int*) – field length ??? see dxf-documentation
- **height** (*float*) – textheight in drawing units (default=1)
- **rotation** (*float*) – text rotation (default=0) (all DXF angles in degrees)
- **oblique** (*float*) – text oblique angle in degree, default=0
- **xscale** (*float*) – width factor (default=1)
- **style** (*str*) – textstyle (default=STANDARD)
- **mirror** (*int*) – bit coded flags
- **halign** (*int*) – horizontal justification type, LEFT, CENTER, RIGHT, ALIGN, FIT, BASELINE_MIDDLE (default LEFT)
- **valign** (*int*) – vertical justification type, TOP, MIDDLE, BOTTOM, BASELINE (default BASELINE)

- **alignpoint** – align point (xy- or xyz-tuple), z-axis is 0 by default, if the justification is anything other than BASELINE/LEFT, alignpoint specify the alignment point (or the second alignment point for ALIGN or FIT).

Flags

flag	description
dxfwrite.ATTRIB_IS_INVISIBLE	Attribute is invisible (does not display)
dxfwrite.ATTRIB_IS_CONST	This is a constant Attribute
dxfwrite.ATTRIB_REQUIRE_VERIFICATION	Verification is required on input of this Attribute
dxfwrite.ATTRIB_IS_PRESET	Verification is required on input of this Attribute

Mirror Flags

flag	description
dxfwrite.MIRROR_X	Text is backward (mirrored in X)
dxfwrite.MIRROR_Y	Text is upside down (mirrored in Y)

Common Keyword Arguments for all Basic DXF R12 Entities

keyword	description
layer	Layer name as string
linetype	Linetype name as string, if not defined = *BYLAYER*
color	as integer in range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*
thickness	Thickness as float
paper_space	0 = entity is in model_space, 1 = entity is in paper_space
extrusion_direction	3D Point as tuple(x, y, z) if extrusion direction is not parallel to the World Z axis

usage:

```
from dxfwrite import DXFEngine as dxf

drawing = dxf.drawing('test.dxf')
block = dxf.block(name='BLOCK1')
attdef = dxf.attdef(insert=(.2, .2),
    rotation=30,
    height=0.25,
    text='test',  # default text
    prompt='input text:', # only important for interactive CAD systems
    tag='BLK')
block.add(attdef)
drawing.block.add(block) # add block definition to drawing

# create a block reference
blockref = dxf.insert(blockname='BLOCK1', insert=(10, 10))

# create a new attribute, given keywords override the default values from
# the attrib definition
attrib = attdef.new_attrib(height=0.18, text='TEST')

# add the attrib to the block reference, insert has the default value (.2, .2),
# and insert is relative to block insert point
blockref.add(attrib, relative=True)
```

```
drawing.add(blockref) # add block reference to drawing
drawing.save()
```

BLOCK

Type: Basic DXF R12 entity.

A block is a collection of objects grouped together to form a single object. You can insert this collection more than once in the same drawing, and when you do, all instances of the block remain identical. You can add fill-in-the-blank text fields, called *attributes*, to blocks.

A block definition lives in an invisible area of your drawing file called the block table. The block table is like a book of graphical recipes for making different kinds of blocks. Each block definition is like a recipe for making one kind of block. To insert a block into a drawing you have to create a block reference by *INSERT*.

You have to add the block definition to the blocks section of the actual drawing:

```
drawing.blocks.add(blockdef)
```

The base point is the point on the block by which you insert it later.

Find a block definition:

```
drawing.blocks.find(blockname)
```

Add entities to a block definition:

```
block.add(entity)
```

DXFEngine.**block**(*name*, *basepoint=(0., 0.)*, ***kwargs*)

Parameters

- **name** (*str*) – blockname
- **basepoint** – block base point (xy- or xyz-tuple), z-axis is 0. by default
- **flags** (*int*) – block type flags
- **xref** (*str*) – xref pathname

where entity can be every drawing entity like circle, line, polyline, attribute, text, ...

Flags

Flags	Description
BLK_ANONYMOUS	This is an anonymous block generated by hatching, associative dimensioning, other internal operations, or an application
BLK_NON_CONSTANT_ATTRIBUTES	This block has non-constant attribute definitions (this bit is not set if the block has any attribute definitions that are constant, or has no attribute definitions at all)
BLK_XREF	This block is an external reference (xref)
BLK_XREF_OVERLAY	This block is an xref overlay
BLK_EXTERNAL	This block is externally dependent
BLK_RESOLVED	This is a resolved external reference, or dependent of an external reference (ignored on input)
BLK_REFERENCED	This definition is a referenced external reference (ignored on input)

Common Keyword Arguments for all Basic DXF R12 Entities

keyword	description
layer	Layer name as string
linetype	Linetype name as string, if not defined = *BYLAYER*
color	as integer in range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*
thickness	Thickness as float
paper_space	0 = entity is in model_space, 1 = entity is in paper_space
extrusion_direction	3D Point as tuple(x, y, z) if extrusion direction is not parallel to the World Z axis

usage:

```
from dxfwrite import DXFEngine as dxf

drawing = dxf.drawing('test.dxf')

# create a block-definition
block = dxf.block(name='BLOCK1')

# add block-definition to drawing
drawing.blocks.add(block)

# create a block-reference
blockref = dxf.insert(blockname='BLOCK1', insert=(10, 10))

# add block-reference to drawing
drawing.add(blockref)

drawing.save()
```

CIRCLE

Type: Basic DXF R12 entity.

A simple circle.

DXFEngine.**circle**(*radius=1.0*, *center=(0., 0.)*, ***kwargs*)

Parameters

- **radius** (*float*) – circle radius
- **center** – center point (xy- or xyz-tuple), z-axis is 0 by default

Common Keyword Arguments for all Basic DXF R12 Entities

keyword	description
layer	Layer name as string
linetype	Linetype name as string, if not defined = *BYLAYER*
color	as integer in range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*
thickness	Thickness as float
paper_space	0 = entity is in model_space, 1 = entity is in paper_space
extrusion_direction	3D Point as tuple(x, y, z) if extrusion direction is not parallel to the World Z axis

Attribs of DXF entities can be changed by the index operator:

```
from dxfwrite import DXFEngine as dxf

drawing = dxf.drawing('drawing.dxf')
circle = dxf.circle(2.0, (1.0, 1.0))
circle['layer'] = 'points'
circle['color'] = 7
circle['center'] = (2, 3, 7) # int or float
circle['radius'] = 3.5
drawing.add(circle)
drawing.save()
```

FACE3D (3DFACE)

Type: Basic DXF R12 entity.

A 3DFace of three or four points.

DXFEngine.**face3d**(*points=[]*, ***kwargs*)

Parameters

- **points** – list of three or four 2D- or 3D-points
- **flags** (*int*) – edge flags, bit-coded, default=0

access/assign 3dface points by index 0, 1, 2 or 3:

```
face3d[0] = (1.2, 4.3, 3.3)
face3d[1] = (7.2, 2.3, 4.4)
```

Flags defined in dxfwrite.const

Name	Value
FACE3D_FIRST_EDGE_IS_INVISIBLE	1
FACE3D_SECOND_EDGE_IS_INVISIBLE	2
FACE3D_THIRD_EDGE_IS_INVISIBLE	4
FACE3D_FOURTH_EDGE_IS_INVISIBLE	8

Common Keyword Arguments for all Basic DXF R12 Entities

keyword	description
layer	Layer name as string
linetype	Linetype name as string, if not defined = *BYLAYER*
color	as integer in range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*
thickness	Thickness as float
paper_space	0 = entity is in model_space, 1 = entity is in paper_space
extrusion_direction	3D Point as tuple(x, y, z) if extrusion direction is not parallel to the World Z axis

Attribs of DXF entities can be changed by the index operator:

```
from dxfwrite import DXFEngine as dxf

drawing = dxf.drawing('drawing.dxf')

# first edge is invisible
```

```
face3d = dxf.face3d([(0, 0), (2, 0), (2, 1), (0, 1)], flags=1)
face3d['layer'] = 'faces'
face3d['color'] = 7

# assign points by index 0, 1, 2, 3
face3d[0] = (1.2, 4.3, 1.9)
drawing.add(face3d)
drawing.save()
```

INSERT

Type: Basic DXF R12 entity.

Insert a new block-reference, for block definitions see *BLOCK*.

DXFEngine.**insert**(*blockname*, *insert=(0., 0.)*, ***kwargs*)

Parameters

- **blockname** (*str*) – name of block definition
- **insert** – insert point (xy- or xyz-tuple), z-axis is 0 by default
- **xscale** (*float*) – x-scale factor, default=1.
- **yscale** (*float*) – y-scale factor, default=1.
- **zscale** (*float*) – z-scale factor, default=1.
- **rotation** (*float*) – rotation angle in degree, default=0.
- **columns** (*int*) – column count, default=1
- **rows** (*int*) – row count, default=1
- **colspacing** (*float*) – column spacing, default=0.
- **rowspacing** (*float*) – row spacing, default=0.

Common Keyword Arguments for all Basic DXF R12 Entities

keyword	description
layer	Layer name as string
linetype	Linetype name as string, if not defined = *BYLAYER*
color	as integer in range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*
thickness	Thickness as float
paper_space	0 = entity is in model_space, 1 = entity is in paper_space
extrusion_direction	3D Point as tuple(x, y, z) if extrusion direction is not parallel to the World Z axis

usage:

```
from dxfwrite import DXFEngine as dxf

drawing = dxf.drawing('test.dxf')
block = dxf.block(name='BLOCK1') # create a block-definition
drawing.block.add(block) # add block-definition to drawing
blockref = dxf.insert(blockname='BLOCK1', insert=(10, 10)) # create a block-reference
drawing.add(blockref) # add block-reference to drawing
drawing.save()
```

LINE

Type: Basic DXF R12 entity.

Draw a single line segment from start point to end point.

DXFEngine.**line**(*start=(0., 0.)*, *end=(0., 0.)*, ***kwargs*)

Parameters

- **start** – start point (xy- or xyz-tuple)
- **end** – end point (xy- or xyz-tuple)

Common Keyword Arguments for all Basic DXF R12 Entities

keyword	description
layer	Layer name as string
linetype	Linetype name as string, if not defined = *BYLAYER*
color	as integer in range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*
thickness	Thickness as float
paper_space	0 = entity is in model_space, 1 = entity is in paper_space
extrusion_direction	3D Point as tuple(x, y, z) if extrusion direction is not parallel to the World Z axis

Attribs of DXF entities can be changed by the index operator:

```
from dxfwrite import DXFEngine as dxf

drawing = dxf.drawing('drawing.dxf')
line = dxf.line((1.2, 3.7), (5.5, 9.7))
line['layer'] = 'walls'
line['color'] = 7
line['start'] = (1.2, 4.3, 1.9)
drawing.add(line)
drawing.save()
```

POINT

Type: Basic DXF R12 entity.

A point simply marks a coordinate. Points are generally used for reference.

DXFEngine.**point**(*point=(0., 0.)*, ***kwargs*)

Parameters

- **point** – start point (xy- or xyz-tuple)
- **orientation** – a 3D vector (xyz-tuple), orientation of PDMODE images ... see dxf documentation

Common Keyword Arguments for all Basic DXF R12 Entities

keyword	description
layer	Layer name as string
linetype	Linetype name as string, if not defined = *BYLAYER*
color	as integer in range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*
thickness	Thickness as float
paper_space	0 = entity is in model_space, 1 = entity is in paper_space
extrusion_direction	3D Point as tuple(x, y, z) if extrusion direction is not parallel to the World Z axis

Attribs of DXF entities can be changed by the index operator:

```
from dxfwrite import DXFEngine as dxf

drawing = dxf.drawing('drawing.dxf')
point = dxf.point((1.0, 1.0))
point['layer'] = 'points'
point['color'] = 7
point['point'] = (2, 3) # int or float
drawing.add(point)
drawing.save()
```

POLYLINE

A polyline is a single object that consists of one or (more usefully) multiple linear segments. You can create open or closed regular or irregular polylines.

POLYMESH and *POLYFACE* are also POLYLINE objects.

Polylines are always 3D-polylines, 2D-polylines are not directly supported, but you can modify the created polylines by clearing the flag *POLYLINE_3D_POLYLINE* to get a 2D polyline.

DXFEngine.**polyline**(*points=[]*, ***kwargs*)

Parameters

- **points** – list of points, 2D or 3D points, z-value of 2D points is 0.
- **polyline_elevation** – polyline elevation (xyz-tuple), z-axis supplies elevation, x- and y-axis has to be 0.)
- **flags** (*int*) – polyline flags, bit-coded, default=0
- **startwidth** (*float*) – default starting width, default=0
- **endwidth** (*float*) – default ending width, default=0
- **mcount** (*int*) – polygon mesh M vertex count, default=0
- **ncount** (*int*) – polygon mesh N vertex count, default=0
- **msmooth_density** (*int*) – (if flags-bit POLYLINE_3D_POLYMESH is set) smooth surface M density, default=0
- **nsmooth_density** (*int*) – (if flags-bit POLYLINE_3D_POLYMESH is set) smooth surface N density, default=0 same values as msmooth_density
- **smooth_surface** (*int*) – curves and smooth surface type, default=0 ??? see dxf-documentation

Common Keyword Arguments for all Basic DXF R12 Entities

keyword	description
layer	Layer name as string
linetype	Linetype name as string, if not defined = *BYLAYER*
color	as integer in range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*
thickness	Thickness as float
paper_space	0 = entity is in model_space, 1 = entity is in paper_space
extrusion_direction	3D Point as tuple(x, y, z) if extrusion direction is not parallel to the World Z axis

Flags

Flag	Description
POLYLINE_CLOSED	This is a closed Polyline (or a polygon mesh closed in the M direction)
POLYLINE_MESH_CLOSED_M_DIRECTION	equals *POLYLINE_CLOSED*
POLYLINE_CURVE_FIT_VERTICES_ADDED	Curve-fit vertices have been added
POLYLINE_SPLINE_FIT_VERTICES_ADDED	Spline-fit vertices have been added
POLYLINE_3D_POLYLINE	This is a 3D Polyline
POLYLINE_3D_POLYMESH	This is a 3D polygon mesh
POLYLINE_MESH_CLOSED_N_DIRECTION	The polygon mesh is closed in the N direction
POLYLINE_POLYFACE_MESH	This Polyline is a polyface mesh
POLYLINE_GENERATE_LINETYPE_PATTERN	The linetype pattern is generated continuously around the vertices of this Polyline

Smooth Density Flags

Flag	Description
POLYMESH_NO_SMOOTH	no smooth surface fitted
POLYMESH_QUADRIC_BSPLINE	quadratic B-spline surface
POLYMESH_CUBIC_BSPLINE	cubic B-spline surface
POLYMESH_BEZIER_SURFACE	Bezier surface

Methods

`Polyline.`**`add_vertex`**(*point*, ***kwargs*)
Add a point to polyline.

Parameters **`point`** – is a 2D or 3D point, z-value of a 2D point is 0.

`Polyline.`**`add_vertices`**(*points*)
Add multiple points.

Parameters **`points`** – list of points, 2D or 3D points, z-value of 2D points is 0.

`Polyline.`**`close`**(*status=True*)
Close Polyline: first vertex is connected with last vertex.

Parameters **status** (*bool*) – *True* close polyline; *False* open polyline

Example:

```
from dxfwrite import DXFEngine as dxf

polyline= dxf.polyline(linetype='DOT')
polyline.add_vertices( [(0,20), (3,20), (6,23), (9,23)] )
drawing.add(polyline)
drawing.save()
```

POLYMESH

Create a new m x n - polymesh entity, polymesh is a dxf-polyline entity!

DXFEngine.**polymesh**(*nrows*, *ncols*, ***kwargs*)

Create a new polymesh entity.

nrows and ncols >=2 and <= 256, greater meshes have to be divided into smaller meshes.

The flags-bit **POLYLINE_3D_POLYMESH** is set.

Parameters

- **nrows** (*int*) – count of vertices in m-direction, nrows >=2 and <= 256
- **ncols** (*int*) – count of vertices in n-direction, ncols >=2 and <= 256

for **kwargs** see *POLYLINE*

Methods

Polymesh.**set_vertex**(*row*, *col*, *point*)

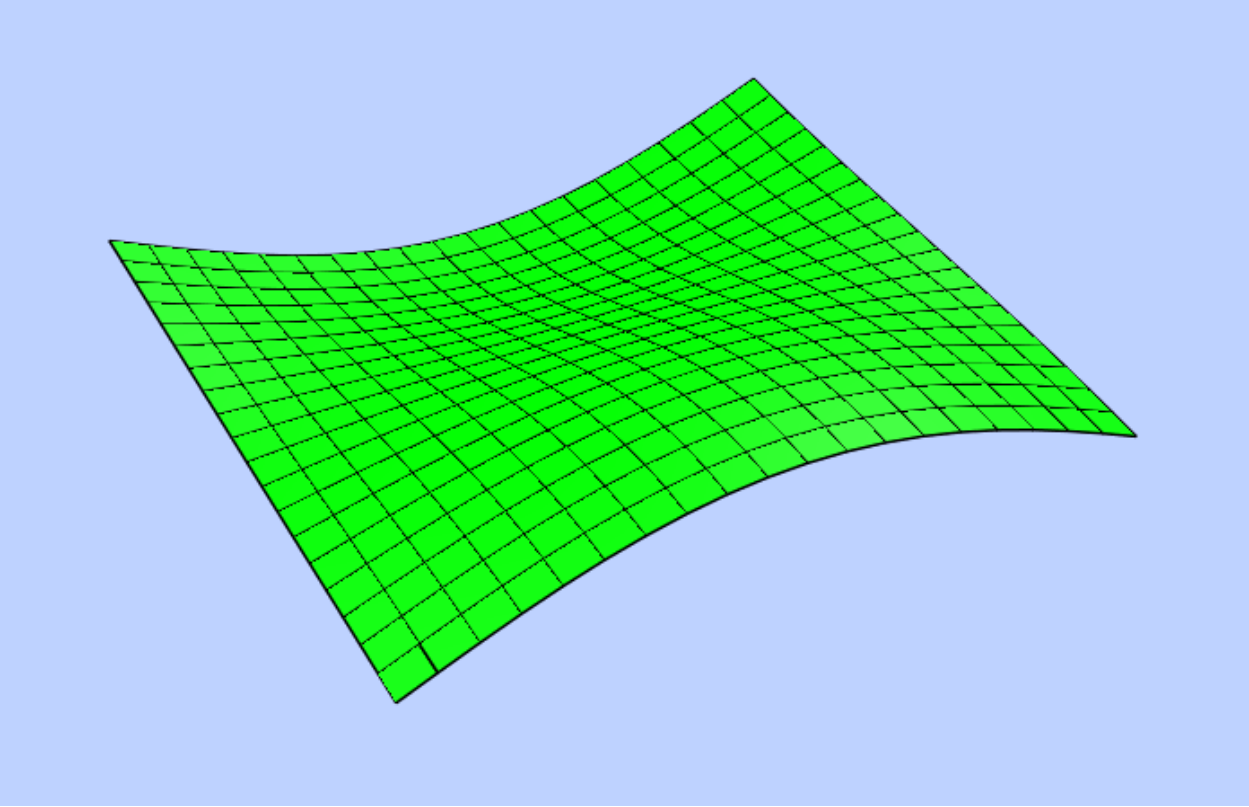
row and col are zero-based indices, point is a tuple (x,y,z)

Polymesh.**set_mclosed**(*status*)

Polymesh.**set_nclosed**(*status*)

Example:

```
import math
from dxfwrite import DXFEngine as dxf

msize, nsize = (20, 20)
dwg = dxf.drawing('mesh.dxf')
mesh = dxf.polymesh(msize, nsize)
delta = math.pi / msize
for x in range(msize):
    sinx = math.sin(float(x)*delta)
    for y in range(nsize):
        cosy = math.cos(float(y)*delta)
        z = sinx * cosy * 3.0
        mesh.set_vertex(x, y, (x, y, z))
dwg.add(mesh)
dwg.save()
```



POLYFACE

Create a new polyface entity, polyface is a dxf-polyline entity!. A Polyface consist of one or more faces, where each face can have three or four 3D points.

DXFEngine.**polyface**(*precision=6*, ***kwargs*)

Parameters **precision** – vertex-coords will be rounded to precision places, and if the vertex is equal to an other vertex, only one vertex will be used, this reduces filespace, the coords will be rounded only for the comparison of the vertices, the output file has the full float resolution.

The flags-bit **POLYLINE_POLYFACE** is set.

for **kwargs** see *POLYLINE*

Methods

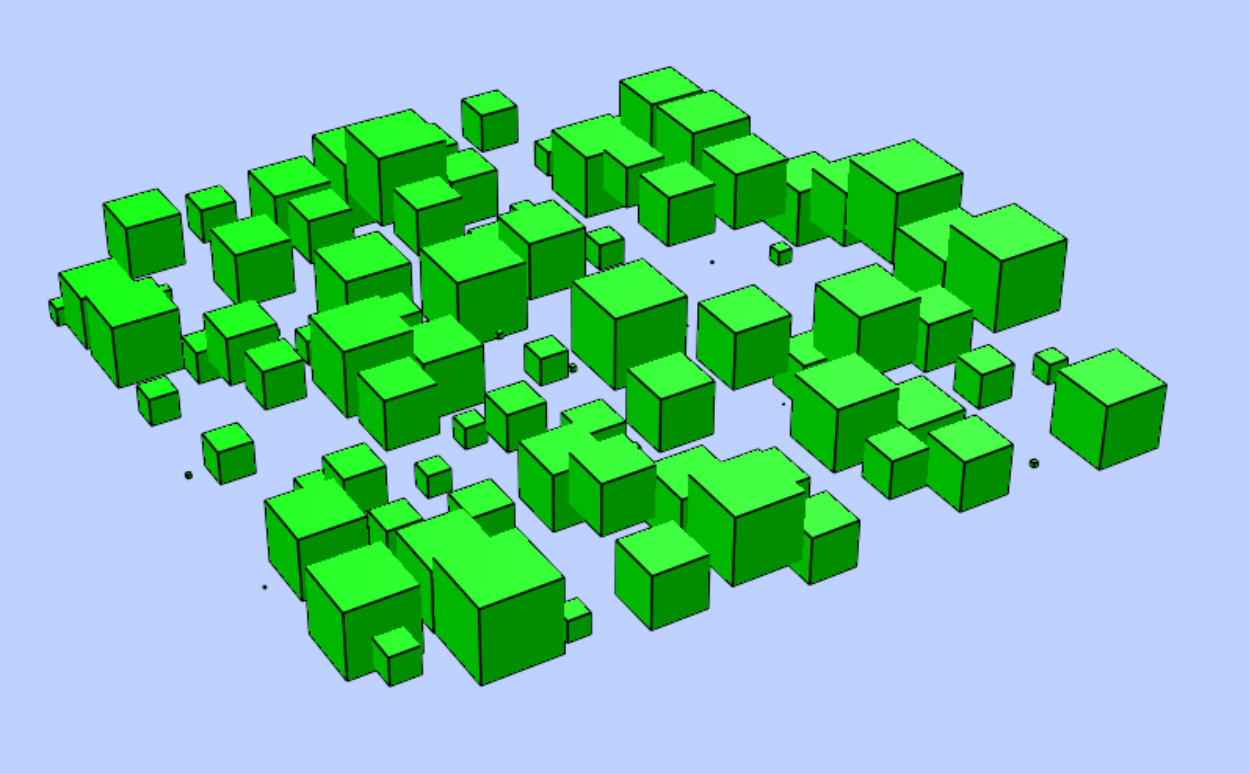
Polyface.add_face(self, vertices, color=0):
This is the recommend method for adding faces.

Parameters

- **vertices** – is a list or tuple with 3 or 4 points (x,y,z).
- **color** (*int*) – range [1..255], 0 = **BYBLOCK**, 256 = **BYLAYER**

Example

```
from dxfwrite import DXFEngine as dxf

def get_cube(basepoint, length):
    def scale( point ):
        return ( (basepoint[0]+point[0]*length),
                 (basepoint[1]+point[1]*length),
                 (basepoint[2]+point[2]*length))
    pface = dxf.polyface()
    # cube corner points
    p1 = scale( (0,0,0) )
    p2 = scale( (0,0,1) )
    p3 = scale( (0,1,0) )
    p4 = scale( (0,1,1) )
    p5 = scale( (1,0,0) )
    p6 = scale( (1,0,1) )
    p7 = scale( (1,1,0) )
    p8 = scale( (1,1,1) )

    # define the 6 cube faces
    # look into -x direction
    # Every add_face adds 4 vertices 6x4 = 24 vertices
    # On dxf output double vertices will be removed.
    pface.add_face([p1, p5, p7, p3], color=1) # base
    pface.add_face([p1, p5, p6, p2], color=2) # left
    pface.add_face([p5, p7, p8, p6], color=3) # front
    pface.add_face([p7, p8, p4, p3], color=4) # right
    pface.add_face([p1, p3, p4, p2], color=5) # back
    pface.add_face([p2, p6, p8, p4], color=6) # top
    return pface

def simple_faces():
    pface = dxf.polyface()
    p1 = (0,0,0)
    p2 = (0,1,0)
    p3 = (1,1,0)
    p4 = (1,0,0)

    p5 = (0,0,1)
    p6 = (0,1,1)
    p7 = (1,1,1)
    p8 = (1,0,1)

    pface.add_face([p1, p2, p3, p4]) # base
    pface.add_face([p5, p6, p7, p8]) # top
```



SHAPE

Type: Basic DXF R12 entity. (untested)

DXFEngine.**shape**(*name*, *insert=(0., 0.)*, ***kwargs*)

Parameters

- **name** (*str*) – name of shape
- **insert** – insert point (xy- or xyz-tuple), z-axis is 0 by default
- **xscale** (*float*) – x-scale factor, default=1.
- **rotation** (*float*) – rotation angle in degree, default=0
- **oblique** (*float*) – text oblique angle in degree, default=0

Common Keyword Arguments for all Basic DXF R12 Entities

keyword	description
layer	Layer name as string
linetype	Linetype name as string, if not defined = *BYLAYER*
color	as integer in range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*
thickness	Thickness as float
paper_space	0 = entity is in model_space, 1 = entity is in paper_space
extrusion_direction	3D Point as tuple(x, y, z) if extrusion direction is not parallel to the World Z axis

SOLID

Type: Basic DXF R12 entity.

Solids are solid-filled 2D outline, a solid can have 3 or 4 points.

DXFEngine.solid(points=[], **kwargs):

Parameters **points** (*list*) – three or four 2D- or 3D-points

access/assign solid points by index 0, 1, 2 or 3:

```
solid[0] = (1.2, 4.3, 3.3)
solid[1] = (7.2, 2.3, 4.4)
```

Common Keyword Arguments for all Basic DXF R12 Entities

keyword	description
layer	Layer name as string
linetype	Linetype name as string, if not defined = *BYLAYER*
color	as integer in range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*
thickness	Thickness as float
paper_space	0 = entity is in model_space, 1 = entity is in paper_space
extrusion_direction	3D Point as tuple(x, y, z) if extrusion direction is not parallel to the World Z axis

Attribs of DXF entities can be changed by the index operator:

```
from dxfwrite import DXFEngine as dxf

drawing = dxf.drawing('drawing.dxf')
solid = dxf.solid([(0, 0), (2, 0), (2, 1), (0, 1)], color=1)
solid['layer'] = 'solids'
solid['color'] = 7

# assign points by index 0, 1, 2, 3
solid[0] = (1.2, 4.3, 1.9)
drawing.add(solid)
drawing.save()
```

TRACE

Type: Basic DXF R12 entity.

A trace of three or four points.

DXFEngine.**trace**(*points=[]*, ***kwargs*)

Parameters **points** – list of three or four 2D- or 3D-points

access/assign trace points by index 0, 1, 2 or 3:

```
trace[0] = (1.2, 4.3, 3.3)
trace[1] = (7.2, 2.3, 4.4)
```

keyword	description
layer	Layer name as string
linetype	Linetype name as string, if not defined = *BYLAYER*
color	as integer in range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*
thickness	Thickness as float
paper_space	0 = entity is in model_space, 1 = entity is in paper_space
extrusion_direction	3D Point as tuple(x, y, z) if extrusion direction is not parallel to the World Z axis

Attribs of DXF entities can be changed by the index operator:

```
from dxfwrite import DXFEngine as dxf

drawing = dxf.drawing('drawing.dxf')
trace = dxf.trace([(0, 0), (2, 0), (2, 1), (0, 1)], layer='0')
trace['layer'] = 'trace'
trace['color'] = 7

# assign points by index 0, 1, 2, 3
trace[0] = (1.2, 4.3, 1.9)
drawing.add(trace)
drawing.save()
```

TEXT

Type: Basic DXF R12 entity.

A simple one line text.

DXFEngine.**text**(*text*, *insert=(0., 0.)*, *height=1.0*, ***kwargs*)

Parameters

- **text** (*str*) – the text to display
- **insert** – insert point (xy- or xyz-tuple), z-axis is 0 by default
- **height** (*float*) – text height in drawing-units
- **rotation** (*float*) – text rotation in degree, default=0
- **xscale** (*float*) – text width factor, default=1
- **oblique** (*float*) – text oblique angle in degree, default=0
- **style** (*str*) – text style name, default=STANDARD
- **mirror** (*int*) – text generation flags, bit-coded, default=0
- **halign** (*int*) – horizontal justification type
- **valign** (*int*) – vertical justification type
- **alignpoint** – align point (xy- or xyz-tuple), z-axis is 0 by default If the justification is anything other than BASELINE/LEFT, alignpoint specify the alignment point (or the second alignment point for ALIGN or FIT).

Common Keyword Arguments for all Basic DXF R12 Entities

keyword	description
layer	Layer name as string
linetype	Linetype name as string, if not defined = *BYLAYER*
color	as integer in range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*
thickness	Thickness as float
paper_space	0 = entity is in model_space, 1 = entity is in paper_space
extrusion_direction	3D Point as tuple(x, y, z) if extrusion direction is not parallel to the World Z axis

Mirror Flags

Flag	Description
const.MIRROR_X	Text is backward (mirrored in X)
const.MIRROR_Y	Text is upside down (mirrored in Y)

Attribs of DXF entities can be changed by the index operator:

```
from dxfwrite import DXFEngine as dxf

drawing = dxf.drawing('drawing.dxf')
text = dxf.text('Text', (1.0, 1.0), height=0.7, rotation=45)
text['layer'] = 'TEXT'
text['color'] = 7
drawing.add(text)
drawing.save()
```

Aligned Text

Attention at aligned Text, if the horizontal align parameter *halign* = `CENTER`, `RIGHT`, `ALIGNED`, `FIT`, `BASELINE_MIDDLE` or the vertical align parameter *valign* = `TOP`, `MIDDLE` or `BOTTOM`, the parameter *alignpoint* defines the text insert point (`CENTER`, `TOP`, ...) or the second align point (`FIT`, `ALIGNED`):

```
from dxfwrite import DXFEngine as dxf
from dxfwrite.const import CENTER

drawing = dxf.drawing('drawing.dxf')
drawing.add(dxf.text('aligned Text', halign=CENTER, alignpoint=(10.0, 5.0)))
drawing.save()
```

VIEWPORT (Entity)

A viewport is a window showing a part of the model space.

You can create a single layout viewport that fits the entire layout or create multiple layout viewports in the paper space (layout).

Note: It is important to create layout viewports on their own layer. When you are ready to plot, you can turn off the layer and plot the layout without plotting the boundaries of the layout viewports. *dxfwrite* uses the layer *VIEWPORTS* as default layer for viewports.

DXFEngine.**viewport**(*center_point*, *width*, *height*, ***kwargs*)

Parameters

- **center_point** – center point of viewport in paper space as (x, y, z) tuple
- **width** (*float*) – width of viewport in paper space
- **height** (*float*) – height of viewport in paper space
- **status** (*int*) – 0 for viewport is off, >0 'stacking' order, 1 is highest priority
- **view_target_point** – as (x, y, z) tuple, default value is (0, 0, 0)
- **view_direction_vector** – as (x, y, z) tuple, default value is (0, 0, 0)
- **view_twist_angle** (*float*) – in radians, default value is 0
- **view_height** (*float*) – default value is 1
- **view_center_point** – as (x, y) tuple, default value is (0, 0)
- **perspective_lens_length** (*float*) – default value is 50
- **front_clip_plane_z_value** (*float*) – default value is 0
- **back_clip_plane_z_value** (*float*) – default value is 0
- **view_mode** (*int*) – default value is 0
- **circle_zoom** (*int*) – default value is 100
- **fast_zoom** (*int*) – default value is 1
- **ucs_icon** (*int*) – default value is 3
- **snap** (*int*) – default value is 0
- **grid** (*int*) – default value is 0
- **snap_style** (*int*) – default value is 0
- **snap_isopair** (*int*) – default value is 0
- **snap_angle** (*float*) – in degrees, default value is 0
- **snap_base_point** – as (x, y) tuple, default value is (0, 0)
- **snap_spacing** – as (x, y) tuple, default value is (0.1, 0.1)
- **grid_spacing** – as (x, y) tuple, default value is (0.1, 0.1)
- **hidden_plot** (*int*) – default value is 0

View Mode Flags

Flags	Description
VMODE_TURNED_OFF	viewport is turned off if bit is set
VMODE_PERSPECTIVE_VIEW_ACTIVE	viewport is in perspective mode if bit is set
VMODE_FRONT_CLIPPING_ON	front clipping is on if bit is set
VMODE_BACK_CLIPPING_ON	back clipping is on if bit is set
VMODE_UCS_FOLLOW_MODE_ON	???
VMODE_FRONT_CLIP_NOT_AT_EYE	???

Common Keyword Arguments for all Basic DXF R12 Entities

keyword	description
layer	Layer name as string
linetype	Linetype name as string, if not defined = *BYLAYER*
color	as integer in range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*

Model space and paper space units

See also:

Paper space (Layout)

Placing the Viewport

The location of the viewport in paper space is defined by the parameters *center_point*, *width* and *height* defines the size of the viewport, all values in paper space coordinates and units. If viewports are overlapping, the display order is defined by the *status* parameter (stacking order), viewports with *status=2* are covered by viewports with *status=1* (*status=1* is the highest display priority). The viewport is always placed in the paper space by default (*paper_space* parameter is 1), placing in model space is possible, but does not display any content.

The viewport content

The viewport gets the content from the model space, the area to show is defined by the parameter *view_target_point* and *view_height*, because the aspect ratio of the viewport is fixed by the parameter *width* and *height*, there is **no** parameter view_width, all values in model space coordinates and units.

Scaling factor

Calculate the scaling factor by *height* divided by *view_height*, example: display a 50.0m model space area in a 1.0m paper space area => 1.0/50.0 => 0.02. If you want a scaling factor of 1:50 (0.02) and the model space area to display is given, calculate the necessary viewport height by *view_height/50*, this is correct if the model space and the paper space has the same drawing units.

Showing 3D content

- Define the *view_target_point* parameter, this is the point you look at.
- Define the *view_direction_vector*, this is just a direction vector, the real location in space is not important.
- The *view_center_point* shifts the viewport,
- and *view_height* determines the model space area to display in the viewport

Example (see also *examples\viewports_in_paperspace.py*):

```
drawing.add(
    DXFEngine.viewport(
        # location of the viewport in paper space
        center_point=(16, 10),
        # viewport width in paper space
        width=4,
```

```
        # viewport height in paper space
        height=4,
        # the model space point you look at
        view_target_point=(40, 40, 0),
        # view_direction_vector determines the view direction,
        # and it just a VECTOR, the view direction is from the location
        # of view_direction_vector to (0, 0, 0)
        view_direction_vector=(-1, -1, 1),
        # now we have a view plane (viewport) with its origin (0, 0) in
        # the view target point and view_center_point shifts
        # the center of the viewport
        view_center_point=(0, 0),
        view_height=30))
```

CHAPTER 3

Composite Entities

MText

Type: Composite Entity

- Multiline-Text buildup with simple Text-Entities
- Mostly the same kwargs like *TEXT*
- Lines are separated by `'\n'`

Caution: align point is always the insert point, I don't need a second alignpoint because horizontal alignment *FIT*, *ALIGN*, *BASELINE_MIDDLE* is not supported by MText.

`DXFEngine.`**`mtext`**(*text*, *insert*, *linespacing=1.5*, ***kwargs*)

Parameters

- **`text`** (*`str`*) – the text to display
- **`insert`** – insert point (xy- or xyz-tuple), z-axis is 0 by default
- **`linespacing`** (*`float`*) – linespacing in percent of height, 1.5 = 150% = 1+1/2 lines
- **`height`** (*`float`*) – text height in drawing-units
- **`rotation`** (*`float`*) – text rotion in dregree, default=0
- **`xscale`** (*`float`*) – text width factor, default=1
- **`oblique`** (*`float`*) – text oblique angle in degree, default=0
- **`style`** (*`str`*) – text style name, default=STANDARD
- **`mirror`** (*`int`*) – text generation flags, bit-coded, default=0
- **`halign`** (*`int`*) – horizontal justification type
- **`valign`** (*`int`*) – vertical justification type
- **`layer`** (*`str`*) – layer name

- **color** (*int*) – range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*

Mirror Flags

Flag	Description
const.MIRROR_X	Text is backward (mirrored in X)
const.MIRROR_Y	Text is upside down (mirrored in Y)

Properties

MText.**lineheight**
Lineheight in drawing units.

Attributes of composite DXF entities can **not** be changed by the index operator, use normal object attributes instead:

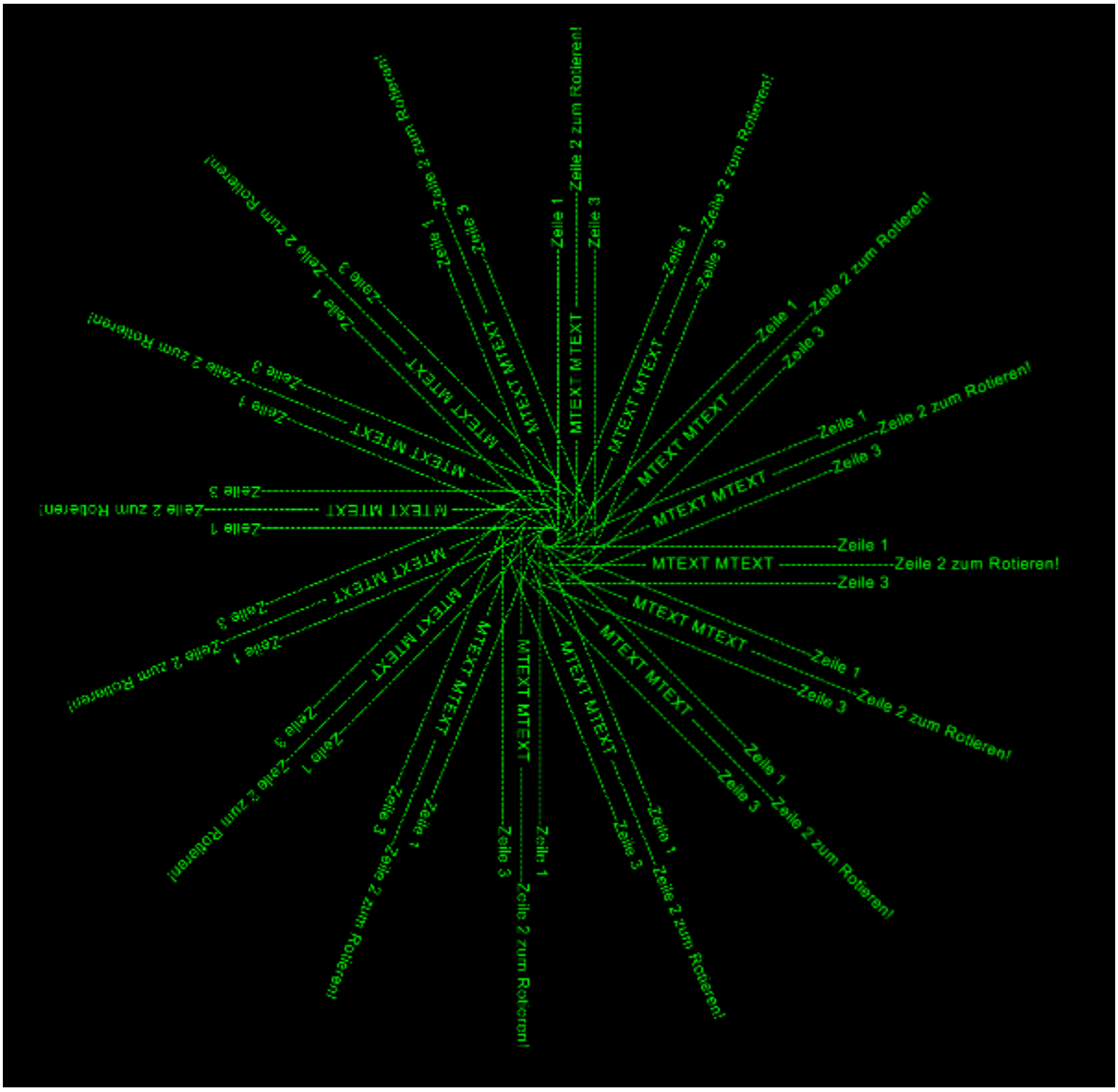
```
from dxfwrite import DXFEngine as dxf

drawing = dxf.drawing('drawing.dxf')
text = dxf.mtext('Line1\nLine2', (1.0, 1.0), height=0.7, rotation=45)
text.layer = 'TEXT'
text.color = 7
drawing.add(text)
drawing.save()
```

Example

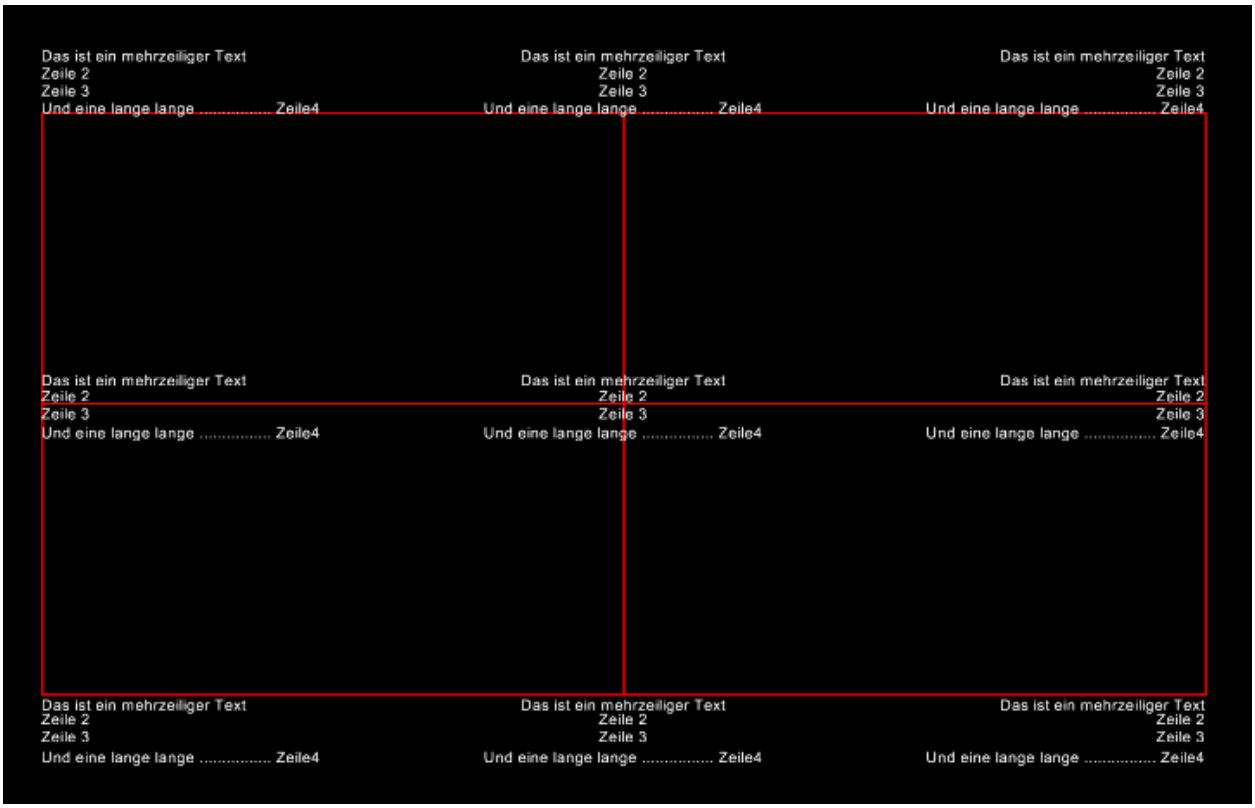
```
import dxfwrite
from dxfwrite import DXFEngine as dxf

def textblock(mtext, x, y, rot, color=3, mirror=0):
    dwg.add(dxf.line((x+50, y), (x+50, y+50),  color=color))
    dwg.add(dxf.line((x+100, y), (x+100, y+50), color=color))
    dwg.add(dxf.line((x+150, y), (x+150, y+50), color=color))

    dwg.add(dxf.line((x+50, y), (x+150, y), color=color))
    dwg.add(dxf.mtext(mtext, (x+50, y), mirror=mirror, rotation=rot))
    dwg.add(dxf.mtext(mtext, (x+100, y), mirror=mirror, rotation=rot,
                      halign=dxfwrite.CENTER))
    dwg.add(dxf.mtext(mtext, (x+150, y), mirror=mirror, rotation=rot,
                      halign=dxfwrite.RIGHT))

    dwg.add(dxf.line((x+50, y+25), (x+150, y+25), color=color))
    dwg.add(dxf.mtext(mtext, (x+50, y+25), mirror=mirror, rotation=rot,
                      valign=dxfwrite.MIDDLE))
    dwg.add(dxf.mtext(mtext, (x+100, y+25), mirror=mirror, rotation=rot,
                      valign=dxfwrite.MIDDLE, halign=dxfwrite.CENTER))
    dwg.add(dxf.mtext(mtext, (x+150, y+25), mirror=mirror, rotation=rot,
                      valign=dxfwrite.MIDDLE, halign=dxfwrite.RIGHT))

    dwg.add(dxf.line((x+50, y+50), (x+150, y+50), color=color))
    dwg.add(dxf.mtext(mtext, (x+50, y+50), mirror=mirror,
                      valign=dxfwrite.BOTTOM, rotation=rot))
    dwg.add(dxf.mtext(mtext, (x+100, y+50), mirror=mirror,
                      valign=dxfwrite.BOTTOM, rotation=rot,
```

```
                        halign=dxfwrite.CENTER))
    dwg.add(dxf.mtext(mtext, (x+150, y+50), mirror=mirror,
                        valign=dxfwrite.BOTTOM, rotation=rot,
                        halign=dxfwrite.RIGHT))

def rotate_text(text, insert, parts=16, color=3):
    delta = 360. / parts
    for part in range(parts):
        dwg.add(dxf.mtext(text, insert, rotation=(delta*part),
                            color=color, valign=dxfwrite.TOP))

name = "mtext.dxf"
dwg = dxf.drawing(name)
txt = "Das ist ein mehrzeiliger Text\nZeile 2\nZeile 3\nUnd eine lange lange" \
        " ................. Zeile4"

textblock(txt, 0, 0, 0., color=1)
textblock(txt, 150, 0,  45., color=2)
textblock(txt, 300, 0,  90., color=3)

textblock(txt, 0, 70,  135., color=4)
textblock(txt, 150, 70,  180., color=5)
textblock(txt, 300, 70,  225., color=6)

txt = "MText Zeile 1\nMIRROR_X\nZeile 3"
textblock(txt, 0, 140,  0., color=4, mirror=dxfwrite.MIRROR_X)
textblock(txt, 150, 140,  45., color=5, mirror=dxfwrite.MIRROR_X)
textblock(txt, 300, 140,  90., color=6, mirror=dxfwrite.MIRROR_X)

txt = "MText Zeile 1\nMIRROR_Y\nZeile 3"
textblock(txt, 0, 210,  0., color=4, mirror=dxfwrite.MIRROR_Y)
textblock(txt, 150, 210,  45., color=5, mirror=dxfwrite.MIRROR_Y)
textblock(txt, 300, 210,  90., color=6, mirror=dxfwrite.MIRROR_Y)

textblock("Einzeiler  0 deg", 0, -70, 0., color=1)
textblock("Einzeiler 45 deg", 150, -70,  45., color=2)
textblock("Einzeiler 90 deg", 300, -70,  90., color=3)

txt = "-------------------------------------------------Zeile 1\n" \
      "----------------- MTEXT MTEXT --------------------Zeile 2 zum Rotieren!\n" \
      "-------------------------------------------------Zeile 3\n"
rotate_text(txt, (600, 100), parts=16, color=3)
dwg.save()
print("drawing '%s' created.\n" % name)
```

Various rotation angles:



Different alignment points:



Insert2

Type: Composite Entity

Insert a new block-reference with auto-creating of *ATTRIB* from *ATTDEF*, and setting attrib-text by the attribs-dict.

See also:

BLOCK, *ATTRIB*, *ATTDEF*, *INSERT*

`DXFEngine.`**`insert2`**(*blockdef*, *insert=(0., 0.)*, *attribs={}*, ***kwargs*)

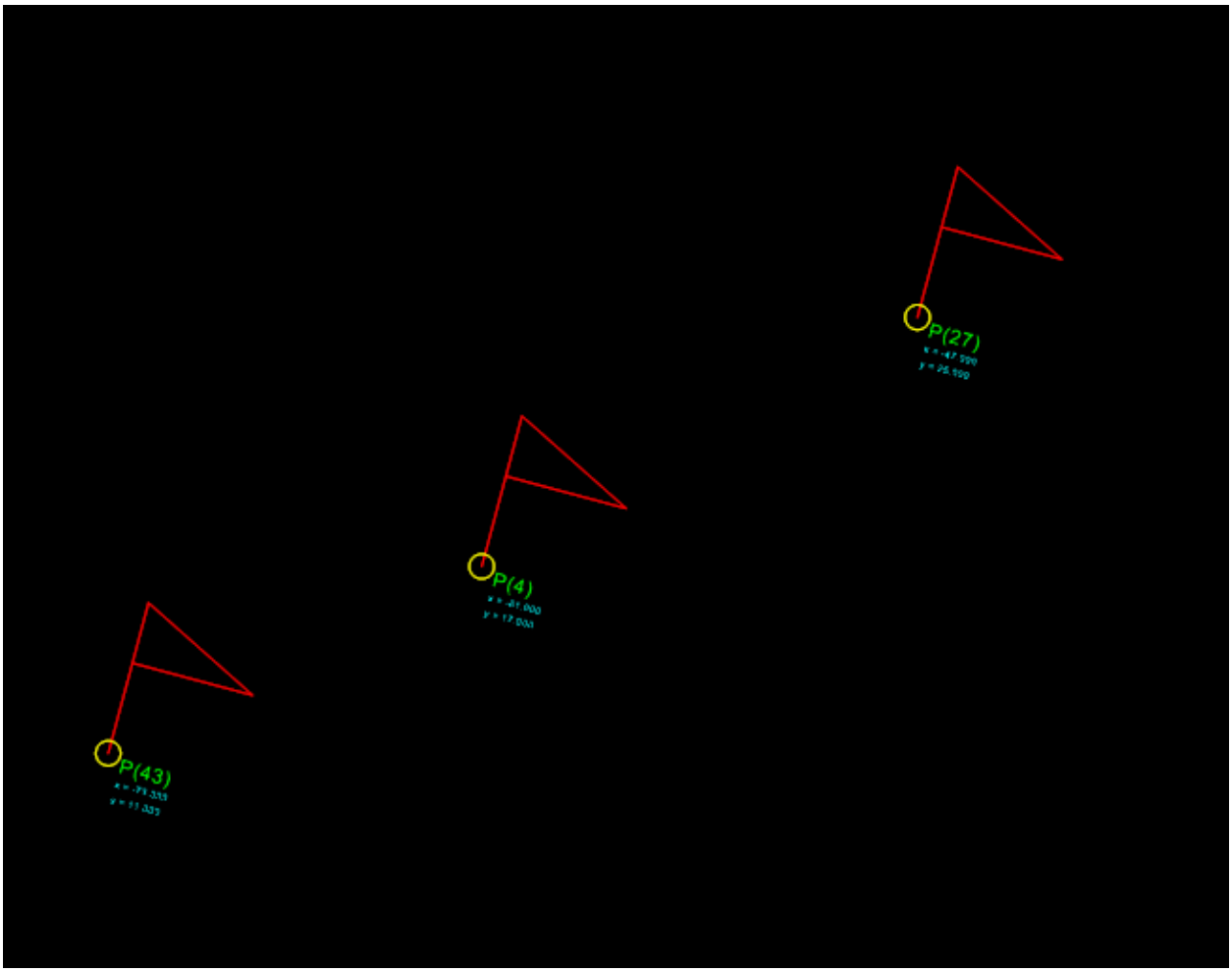
Parameters

- **`blockdef`** – the block definition itself
- **`insert`** – insert point (xy- or xyz-tuple), z-axis is 0 by default
- **`xscale`** (*`float`*) – x-scale factor, default=1.
- **`yscale`** (*`float`*) – y-scale factor, default=1.
- **`zscale`** (*`float`*) – z-scale factor, default=1.
- **`rotation`** (*`float`*) – rotation angle in degree, default=0.
- **`attribs`** (*`dict`*) – dict with tag:value pairs, to fill the the attdefs in the block-definition. example: {'TAG1': 'TextOfTAG1'}, create and insert an attrib from an attdef (with tag-value == 'TAG1'), and set text-value of the attrib to value 'TextOfTAG1'.
- **`linetype`** (*`string`*) – linetype name, if not defined = *BYLAYER*
- **`layer`** (*`string`*) – layer name

- **color** (*int*) – range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*

Example

```
import dxfwrite
from dxfwrite import DXFEngine as dxf

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

sample_coords = [get_random_point() for x in range(50)]

flag_symbol = [(0,0), (0, 5), (4, 3), (0, 3)]

filename = 'flags.dxf'
dwg = dxf.drawing(filename)
dwg.add_layer('FLAGS')

# first create a block
flag = dxf.block(name='flag')
# add dxf entities to the block (the flag)
# use basepoint = (x, y) define an other basepoint than (0, 0)
flag.add( dxf.polyline(flag_symbol) )
flag.add( dxf.circle(radius=.4, color=2) )
# define some attributes
flag.add( dxf.attdef(insert=(0.5, -0.5), tag='NAME', height=0.5, color=3) )
flag.add( dxf.attdef(insert=(0.5, -1.0), tag='XPOS', height=0.25, color=4) )
flag.add( dxf.attdef(insert=(0.5, -1.5), tag='YPOS', height=0.25, color=4) )

# add block definition to the drawing
dwg.blocks.add(flag)
number = 1
for point in sample_coords:
    # now insert flag symbols at coordinate 'point'
    # insert2 needs the block definition object as parameter 'blockdef'
    # see http://packages.python.org/dxfwrite/entities/insert2.html
    # fill attribtes by creating a dict(), keystr is the 'tag' name of the
    # attribute
    values = {
        'NAME': "P(%d)" % number,
        'XPOS': "x = %.3f" % point[0],
        'YPOS': "y = %.3f" % point[1]
    }
    randomscale = 0.5 + random.random() * 2.0
    dwg.add(dxf.insert2(blockdef=flag, insert=point,
                        attribs=values,
                        xscale=randomscale,
                        yscale=randomscale,
                        layer='FLAGS', rotation=-15))
    number += 1

dwg.save()
print("drawing '%s' created.\n" % filename)
```



LinearDimension

Type: Composite Entity

class `LinearDimension`

Simple straight dimension line with two or more measure points, build with basic DXF entities. This is NOT a dxf dimension entity. And This is a 2D element, so all z-values will be ignored!

`LinearDimension.__init__`(*pos*, *measure_points*, *angle=0.*, *dimstyle='Default'*, *layer=None*, *roundval=None*)

Parameters

- **pos** – location as (x, y) tuple of dimension line, line goes through this point
- **measure_points** – list of points as (x, y) tuples to dimension (two or more)
- **angle** (*float*) – angle (in degree) of dimension line
- **dimstyle** (*str*) – dimstyle name, 'Default' - style is the default value
- **layer** (*str*) – dimension line layer, override the default value of dimstyle
- **roundval** (*int*) – count of decimal places

Methods

LinearDimension.**set_text**(*section*, *text*)
Set and override the text of the dimension text for the given dimension line section.

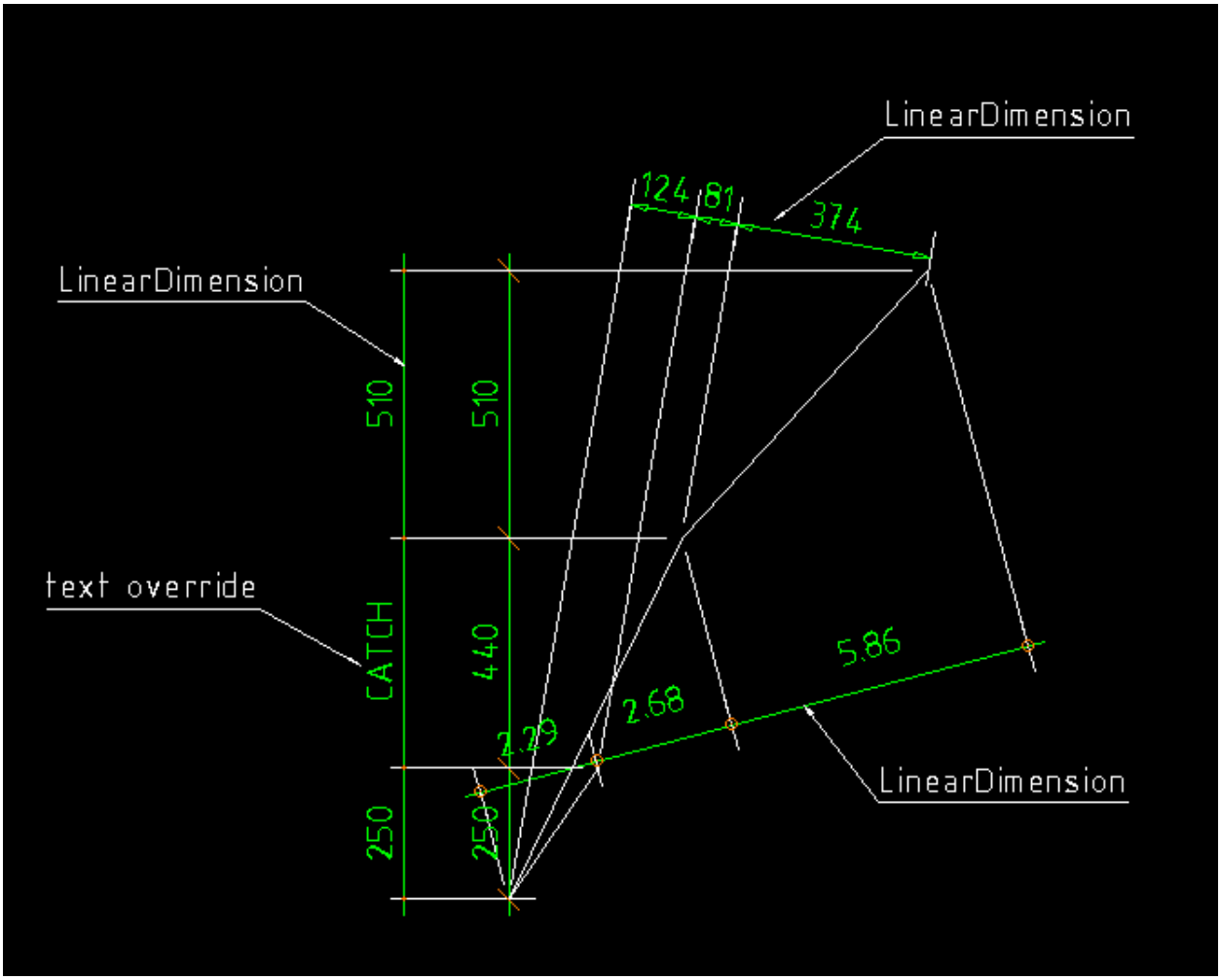
Properties

LinearDimension.**section_count**
count of dimline sections

LinearDimension.**point_count**
count of dimline points

Example

```
import dxfwrite
from dxfwrite import DXFEngine as dxf

# Dimlines are separated from the core library.
# Dimension lines will not generated by the DXFEngine.
from dxfwrite.dimlines import dimstyles, LinearDimension

# create a new drawing
dwg = dxf.drawing('dimlines.dxf')

# dimensionline setup:
# add block and layer definition to drawing
dimstyles.setup(dwg)

# create a dimension line for following points
points = [ (1.7,2.5), (0,0), (3.3,6.9), (8,12)]

# define new dimstyles, for predefined ticks see dimlines.py
dimstyles.new("dots", tick="DIMTICK_DOT", scale=1., roundval=2, textabove=.5)
dimstyles.new("arrow", tick="DIMTICK_ARROW", tick2x=True, dimlineext=0.)
dimstyles.new('dots2', tick="DIMTICK_DOT", tickfactor=.5)

#add linear dimension lines
dwg.add(LinearDimension((3,3), points, dimstyle='dots', angle=15.))
dwg.add(LinearDimension((0,3), points, angle=90.))
dwg.add(LinearDimension((-2,14), points, dimstyle='arrow', angle=-10))

# next dimline is added as anonymous block
dimline = LinearDimension((-2,3), points, dimstyle='dots2', angle=90.)

# override dimension text
dimline.set_text(1, 'CATCH')

# add dimline as anonymous block
dwg.add_anonymous_block(dimline, layer='DIMENSIONS')
```



AngularDimension

Type: Composite Entity

class AngularDimension

Draw an angle dimensioning line at dimline *pos* from *start* to *end*, dimension text is the angle build of the three points start-center-end.

AngularDimension.**__init__**(*pos*, *center*, *start*, *end*, *dimstyle='angle.deg'*, *layer=None*, *roundval=None*)

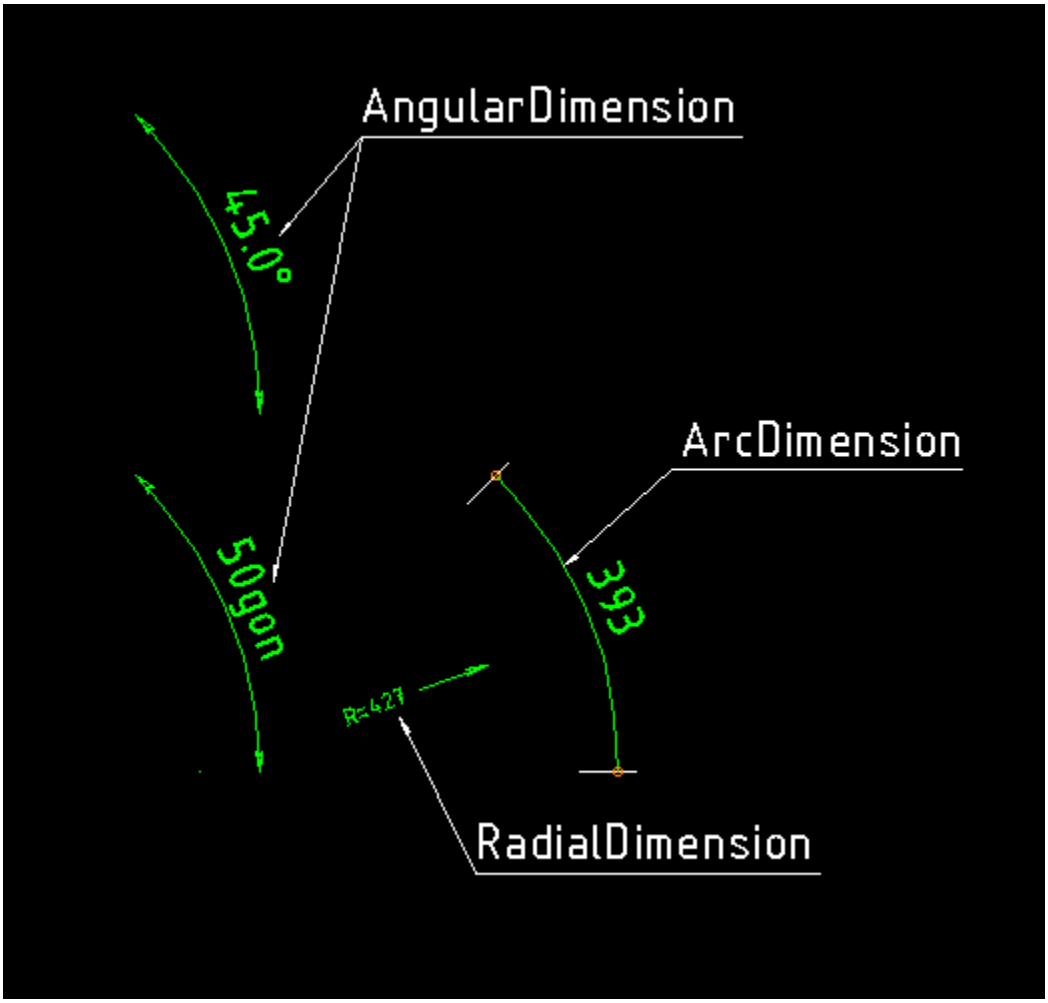
Parameters

- **pos** – location as (x, y) tuple of dimension line, line goes through this point
- **center** – center point as (x, y) tuple of angle
- **start** – line from center to start is the first side of the angle
- **end** – line from center to end is the second side of the angle
- **dimstyle** (*str*) – dimstyle name, 'Default' - style is the default value
- **layer** (*str*) – dimension line layer, override the default value of dimstyle

- **roundval** (*int*) – count of decimal places

Example

```
import dxfwrite
from dxfwrite import DXFEngine as dxf

# Dimlines are separated from the core library.
# Dimension lines will not generated by the DXFEngine.
from dxfwrite.dimlines import dimstyles, AngularDimension

# create a new drawing
dwg = dxf.drawing('dimlines.dxf')

# dimensionline setup:
# add block and layer definition to drawing
dimstyles.setup(dwg)

# There are three dimstyle presets for angular dimension
# 'angle.deg' (default), 'angle.rad', 'angle.grad' (gon)
# for deg and grad default roundval = 0
# for rad default roundval = 3

# angular dimension in grad (gon)
dwg.add(AngularDimension(pos=(18, 5), center=(15, 0), start=(20, 0),
                         end=(20, 5), dimstyle='angle.grad'))

# angular dimension in degree (default dimstyle), with one fractional digit
dwg.add(AngularDimension(pos=(18, 10), center=(15, 5), start=(20, 5),
                         end=(20, 10), roundval=1))
```



ArcDimension

Type: Composite Entity

class ArcDimension

Arc is defined by start- and endpoint on arc and the centerpoint, or by three points lying on the arc if acr3points is True. Measured length goes from start- to endpoint. The dimension line goes through the dimlinepos.

ArcDimension.**__init__**(*pos*, *center*, *start*, *end*, *arc3points=False*, *dimstyle='Default'*, *layer=None*, *roundval=None*)

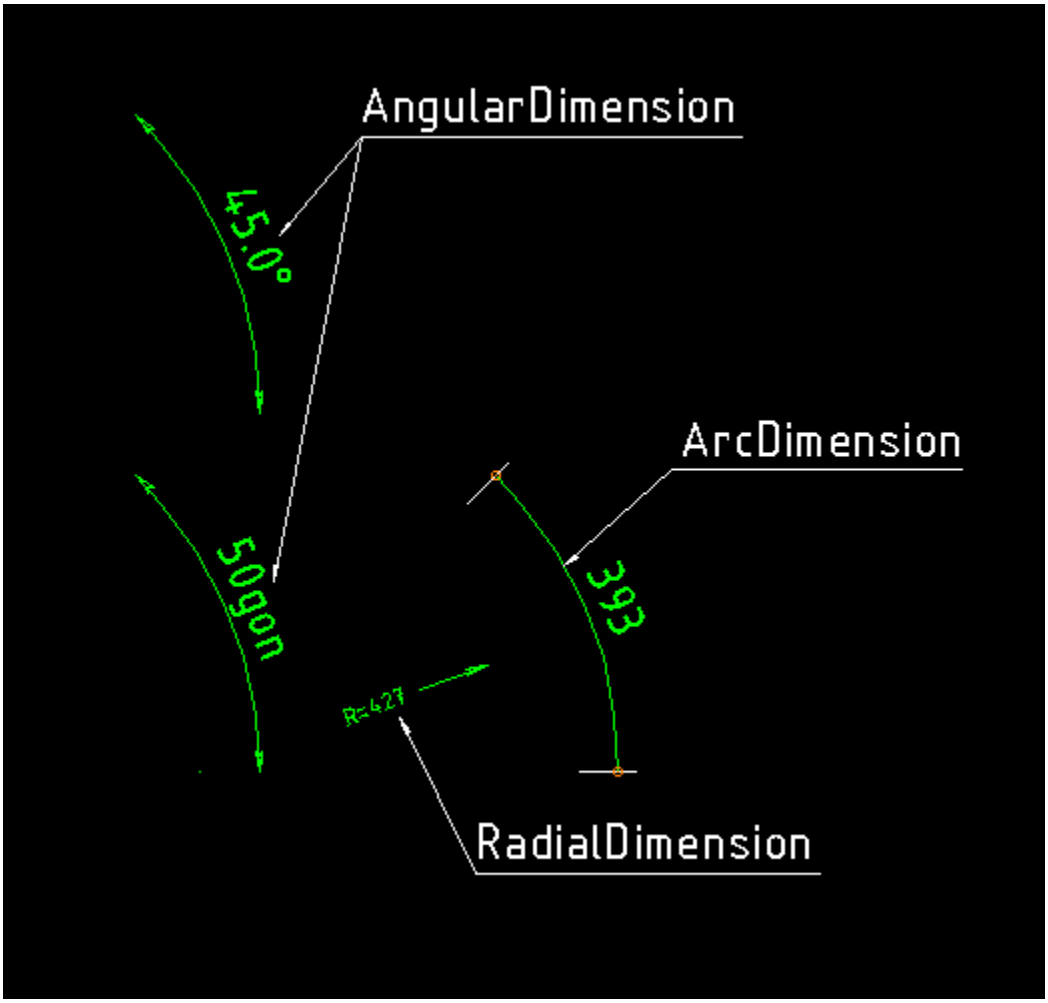
Parameters

- **pos** – location as (x, y) tuple of dimension line, line goes through this point
- **center** – center point of arc
- **start** – start point of arc
- **end** – end point of arc
- **arc3points** (*bool*) – if **True** arc is defined by three points on the arc (center, start, end)
- **dimstyle** (*str*) – dimstyle name, 'Default' - style is the default value

- **layer** (*str*) – dimension line layer, override the default value of dimstyle
- **roundval** (*int*) – count of decimal places

Example

```
import dxfwrite
from dxfwrite import DXFEngine as dxf

# Dimlines are separated from the core library.
# Dimension lines will not generated by the DXFEngine.
from dxfwrite.dimlines import dimstyles, ArcDimension

# create a new drawing
dwg = dxf.drawing('dimlines.dxf')

# dimensionline setup:
# add block and layer definition to drawing
dimstyles.setup(dwg)

# define new dimstyles, for predefined ticks see dimlines.py
dimstyles.new('dots2', tick="DIMTICK_DOT", tickfactor=.5)

dwg.add(ArcDimension(pos=(23, 5), center=(20, 0), start=(25, 0),
                     end=(25, 5), dimstyle='dots2'))
```



RadialDimension

Type: Composite Entity

class `RadialDimension`

Draw a radius dimension line from *target* in direction of *center* with length drawing units. RadialDimension has a special tick!!

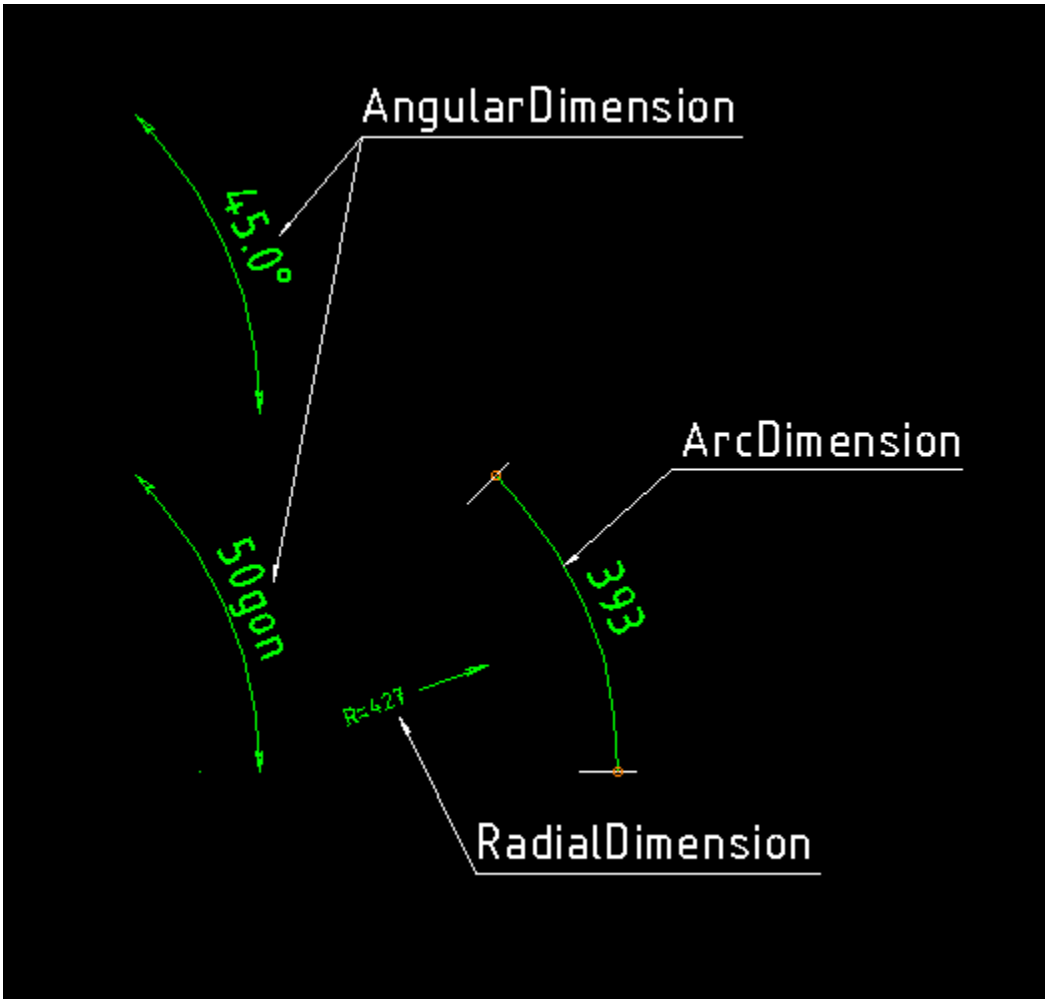
`RadialDimension.__init__`(*center*, *target*, *length=1.*, *dimstyle='Default'*, *layer=None*, *roundval=None*)

Parameters

- **`center`** – center point of radius
- **`target`** – target point of radius
- **`length`** (*`float`*) – length of radius arrow (drawing length)
- **`dimstyle`** (*`str`*) – dimstyle name, 'Default' - style is the default value
- **`layer`** (*`str`*) – dimension line layer, override the default value of dimstyle
- **`roundval`** – count of decimal places

Example

```
import dxfwrite
from dxfwrite import DXFEngine as dxf

# Dimlines are separated from the core library.
# Dimension lines will not generated by the DXFEngine.
from dxfwrite.dimlines import dimstyles, RadialDimension

# create a new drawing
dwg = dxf.drawing('dimlines.dxf')

# dimensionline setup:
# add block and layer definition to drawing
dimstyles.setup(dwg)

# RadialDimension has a special tick
dimstyles.new("radius", height=0.25, prefix='R=')
dwg.add(RadialDimension((20, 0), (24, 1.5), dimstyle='radius'))
```



Rectangle

Type: Composite Entity

2D Rectangle, build with a polyline and a solid as background filling

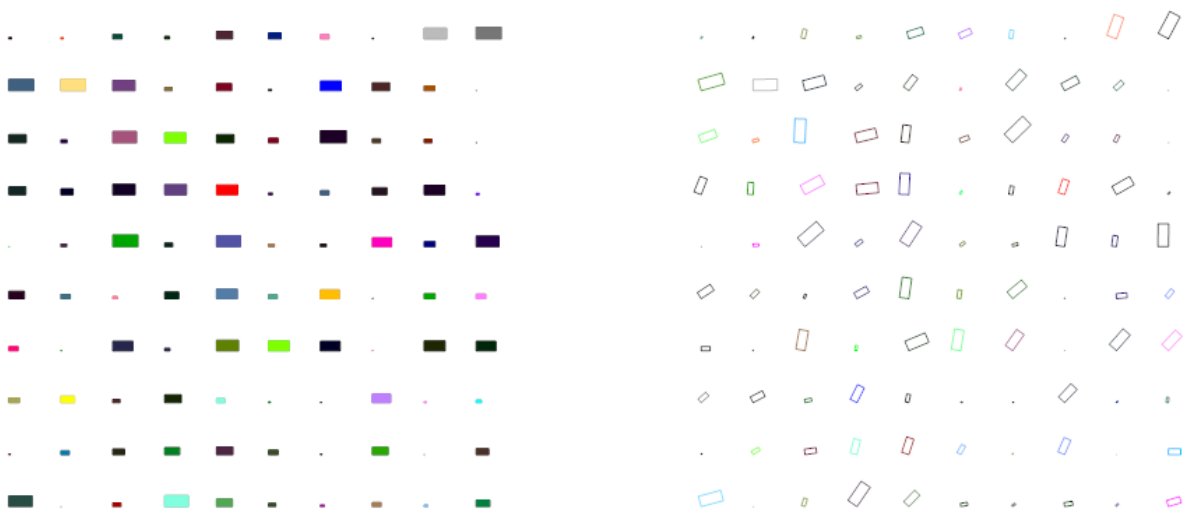
DXFEngine.**rectangle**(*insert*, *width*, *height*, ***kwargs*)

Parameters

- **insert** (*point*) – where to place the rectangle
- **width** (*float*) – width in drawing units
- **height** (*float*) – height in drawing units
- **rotation** (*float*) – in degree (circle = 360 degree)
- **halign** (*int*) – *LEFT*, *CENTER*, *RIGHT*
- **valign** (*int*) – *TOP*, *MIDDLE*, *BOTTOM*
- **color** (*int*) – dxf color index, default is *BYLAYER*, if color is None, no polyline will be created, and the rectangle consist only of the background filling (if bgcolor != *None*)
- **bgcolor** (*int*) – dxf color index, default is *None* (no background filling)
- **layer** (*string*) – target layer, default is '0'
- **linetype** (*string*) – linetype name, None = *BYLAYER*

Example

```
import dxfwrite
from dxfwrite import DXFEngine as dxf

name="rectangle.dxf"
drawing = dxf.drawing(name)

for x in range(10):
    for y in range(10):
        color = 255 * random()
        bgcolor = 255 * random()
        rand = random()
        # rectangle with only backgound filling
        drawing.add(dxf.rectangle((x*3, y*3) , 1.5*rand, .7*rand,
                                  bgcolor=bgcolor))
        angle = 90 * random()
        # rectangle with only border lines
        drawing.add(dxf.rectangle((40+(x*3), y*3) , 1.5*rand, .7*rand,
                                  color=color, rotation=angle))
drawing.save()
print("drawing '%s' created.\n" % name)
```



Table

Type: Composite Entity

Table object like a HTML-Table, buildup with basic DXF R12 entities.

Cells can contain Multiline-Text or DXF-BLOCKs, or you can create your own cell-type by extending the CustomCell object.

Cells can span over columns and rows.

Text cells can contain text with an arbitrary rotation angle, or letters can be stacked top-to-bottom.

BlockCells contains block references (INSERT-entity) created from a block definition (BLOCK), if the block definition contains attribute definitions (ATTDEF-entity), attribs created by Attdef.new_attrib() will be added to the block reference (ATTRIB-entity).

`DXFEngine.`**`table`**(*insert*, *nrows*, *ncols*, *default_grid=True*)

Parameters

- **`insert`** – insert point as 2D or 3D point
- **`nrows`** (*int*) – row count
- **`ncols`** (*int*) – column count
- **`default_grid`** (*bool*) – if *True* always a solid line grid will be drawn, if *False*, only explicit defined borders will be drawn, default grid has a priority of 50.

Methods

`Table.`**`set_col_width`**(*column*, *value*)

Set width of 'column' to 'value'.

Parameters

- **`column`** (*int*) – zero based column index

- **value** (*float*) – new column width in drawing units

Table.**set_row_height**(*row*, *value*)
Set height of 'row' to 'value'.

Parameters

- **row** (*int*) – zero based row index
- **value** (*float*) – new row height in drawing units

Table.**text_cell**(*row*, *col*, *text*, *span=(1, 1)*, *style='default'*)
Create a new text cell at position (row, col), with 'text' as content, text can be a multi-line text, use `'\\n'` as line separator.

The cell spans over *span* cells and has the cell style with the name *style*.

Table.**block_cell**(*row*, *col*, *blockdef*, *span=(1, 1)*, *attribs={}*, *style='default'*)
Create a new block cell at position (row, col).

Content is a block reference inserted by a *INSERT* entity, attributes will be added if the block definition contains *ATTDEF*. Assignments are defined by attribs-key to attdef-tag association.

Example: attribs = {'num': 1} if an *ATTDEF* with tag=='num' in the block definition exists, an attrib with text=str(1) will be created and added to the insert entity.

The cell spans over 'span' cells and has the cell style with the name 'style'.

Table.**set_cell**(*row*, *col*, *cell*)
Insert a cell at position (row, col).

Table.**get_cell**(*row*, *col*)
Get cell at position (row, col).

Table.**frame**(*row*, *col*, *width=1*, *height=1*, *style='default'*)
Create a Frame object which frames the cell area starting at (row, col), covering 'width' columns and 'height' rows.

Table.**new_cell_style**(*name*, ***kwargs*)
Create a new Style object 'name'.

The 'kwargs' are the key, value pairs of of the dict *Cellstyle*.

Table.new_border_style(color=const.BYLAYER, status=True, priority=100, linetype=None):
Create a new border style.

Parameters

- **status** (*bool*) – True for visible, else False
- **color** (*int*) – dxf color index
- **linetype** (*str*) – linetype name, BYLAYER if None
- **priority** (*int*) – drawing priority - higher values covers lower values

see also *Borderstyle*.

Table.**get_cell_style**(*name*)
Get cell style by name.

Table.**iter_visible_cells**()
Iterate over all visible cells.

Returns a generator which yields all visible cells as tuples: (row , col, cell)

Cellstyle

Just a python dict:

```
{
    # textstyle is ignored by block cells
    'textstyle': 'STANDARD',
    # text height in drawing units, ignored by block cells
    'textheight': DEFAULT_CELL_TEXT_HEIGHT,
    # line spacing in percent = <textheight>*<linespacing>, ignored by block cells
    'linespacing': DEFAULT_CELL_LINESPACING,
    # text stretch or block reference x-axis scaling factor
    'xscale': DEFAULT_CELL_XSCALE,
    # block reference y-axis scaling factor, ignored by text cells
    'yscale': DEFAULT_CELL_YSCALE,
    # dxf color index, ignored by block cells
    'textcolor': DEFAULT_CELL_TEXTCOLOR,
    # text or block rotation in degrees
    'rotation' : 0.,
    # Letters are stacked top-to-bottom, but not rotated
    'stacked': False,
    # horizontal alignment (const.LEFT, const.CENTER, const.RIGHT)
    'halign': DEFAULT_CELL_HALIGN,
    # vertical alignment (const.TOP, const.MIDDLE, const.BOTTOM)
    'valign': DEFAULT_CELL_VALIGN,
    # left and right margin in drawing units
    'hmargin': DEFAULT_CELL_HMARGIN,
    # top and bottom margin
    'vmargin': DEFAULT_CELL_VMARGIN,
    # background color, dxf color index, ignored by block cells
    'bgcolor': DEFAULT_CELL_BG_COLOR,
    # left border style
    'left': Style.get_default_border_style(),
    # top border style
    'top': Style.get_default_border_style(),
    # right border style
    'right': Style.get_default_border_style(),
    # bottom border style
    'bottom': Style.get_default_border_style(),
}
```

Borderstyle

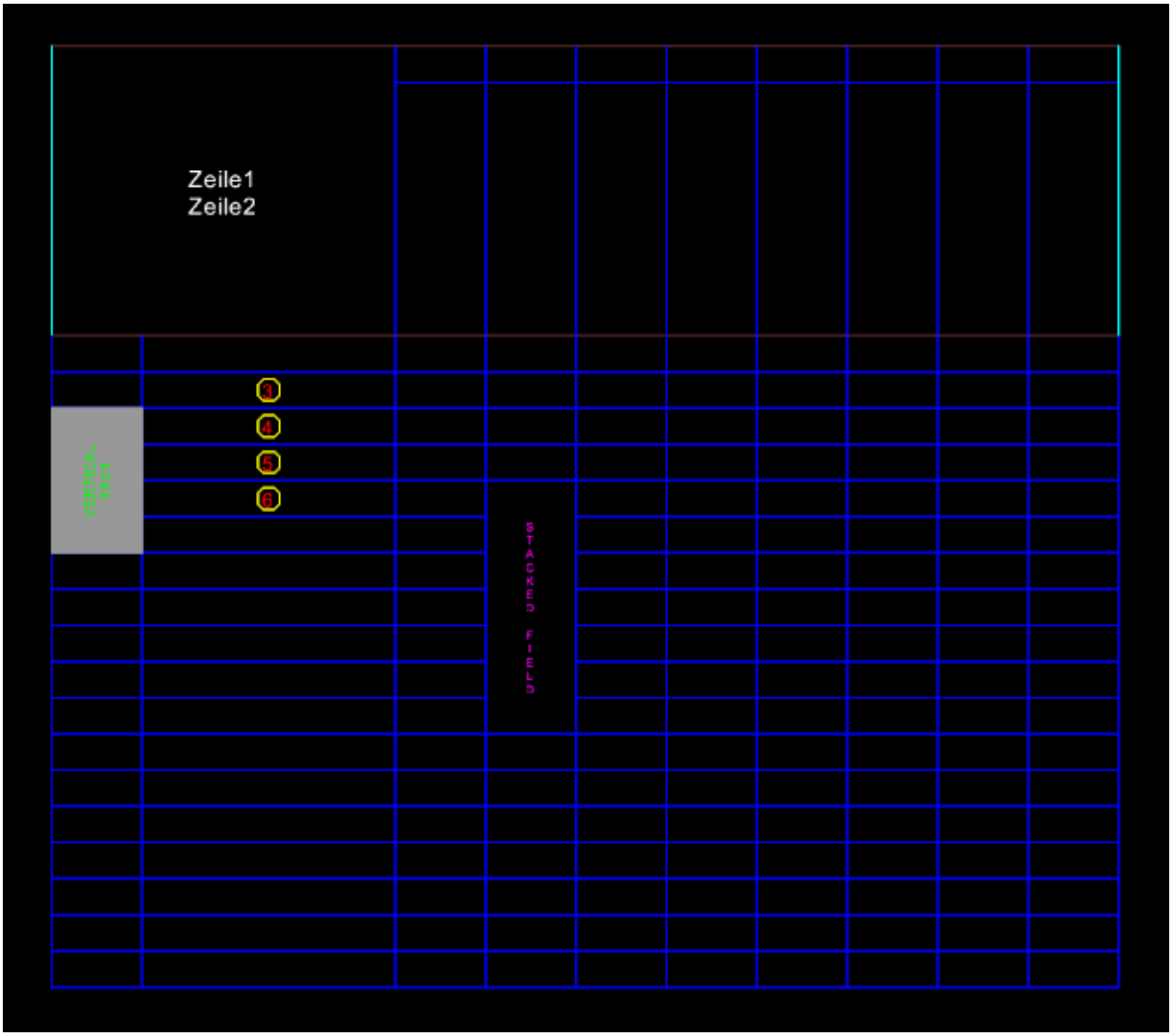
Just a python dict:

```
{
    # border status, True for visible, False for hidden
    'status': DEFAULT_BORDER_STATUS,
    # dxf color index
    'color': DEFAULT_BORDER_COLOR,
    # linetype name, BYLAYER if None
    'linetype': DEFAULT_BORDER_LINETYPE,
    # drawing priority, higher values cover lower values
    'priority': DEFAULT_BORDER_PRIORITY,
}
```

Example

```
import dxfwrite
from dxfwrite import DXFEngine as dxf

def get_mat_symbol():
    p1 = 0.5
    p2 = 0.25
    points = [(p1, p2), (p2, p1), (-p2, p1), (-p1, p2), (-p1, -p2),
              (-p2, -p1), (p2, -p1), (p1, -p2)]
    polygon = dxf.polyline(points, color=2)
    polygon.close()
    attdef = dxf.attdef(text='0', tag='num', height=0.7, color=1,
                        halign=dxfwrite.CENTER, valign=dxfwrite.MIDDLE
                        )
    symbolblock = dxf.block('matsymbol')
    symbolblock.add(polygon)
    symbolblock.add(attdef)
    dwg.blocks.add(symbolblock)
    return symbolblock

name = 'table.dxf'
dwg = dxf.drawing(name) # create a drawing
table = dxf.table(insert=(0, 0), nrows=20, ncols=10)
# create a new styles
ctext = table.new_cell_style('ctext', textcolor=7, textheight=0.5,
                             halign=dxfwrite.CENTER,
                             valign=dxfwrite.MIDDLE
                             )
# modify border settings
border = table.new_border_style(color=6, linetype='DOT', priority=51)
ctext.set_border_style(border, right=False)

table.new_cell_style('vtext', textcolor=3, textheight=0.3,
                     rotation=90, # vertical written
                     halign=dxfwrite.CENTER,
                     valign=dxfwrite.MIDDLE,
                     bgcolor=8,
                     )
# set colum width, first column has index 0
table.set_col_width(1, 7)

#set row height, first row has index 0
table.set_row_height(1, 7)

# create a text cell with the default style
cell1 = table.text_cell(0, 0, 'Zeile1\nZeile2', style='ctext')

# cell spans over 2 rows and 2 cols
cell1.span=(2, 2)

cell2 = table.text_cell(4, 0, 'VERTICAL\nTEXT', style='vtext', span=(4, 1))

# create frames
table.frame(0, 0, 10, 2, 'framestyle')

# because style is defined by a namestring
# style can be defined later
```

```
hborder = table.new_border_style(color=4)
vborder = table.new_border_style(color=17)
table.new_cell_style('framestyle', left=hborder, right=hborder,
                     top=vborder, bottom=vborder)
mat_symbol = get_mat_symbol()

table.new_cell_style('matsym',
                     halign=dxfwrite.CENTER,
                     valign=dxfwrite.MIDDLE,
                     xscale=0.6, yscale=0.6)

# add table as anonymous block
# dxf creation is only done on save, so all additional table inserts
# which will be done later, also appear in the anonymous block.

dwg.add_anonymous_block(table, insert=(40, 20))

# if you want different tables, you have to deepcopy the table
newtable = deepcopy(table)
newtable.new_cell_style('57deg', textcolor=2, textheight=0.5,
                     rotation=57, # write
                     halign=dxfwrite.CENTER,
                     valign=dxfwrite.MIDDLE,
                     bgcolor=123,
                     )
newtable.text_cell(6, 3, "line one\nline two\nand line three",
                   span=(3,3), style='57deg')
dwg.add_anonymous_block(newtable, basepoint=(0, 0), insert=(80, 20))

# a stacked text: Letters are stacked top-to-bottom, but not rotated
table.new_cell_style('stacked', textcolor=6, textheight=0.25,
                     halign=dxfwrite.CENTER,
                     valign=dxfwrite.MIDDLE,
                     stacked=True)
table.text_cell(6, 3, "STACKED FIELD", span=(7, 1), style='stacked')

for pos in [3, 4, 5, 6]:
    blockcell = table.block_cell(pos, 1, mat_symbol,
                                 attribs={'num': pos},
                                 style='matsym')

dwg.add(table)
dwg.save()
print("drawing '%s' created.\n" % name)
```



Ellipse

Type: Composite Entity

Ellipse curves are approximated by a *POLYLINE*.

For an explanation of ellipse curves see Wikipedia:

http://en.wikipedia.org/wiki/Ellipse

`DXFEngine.`**`ellipse`**(*center*, *rx*, *ry*, *startangle=0.*, *endangle=360.*, *rotation=0.*, *segments=100*, ***kwargs*)

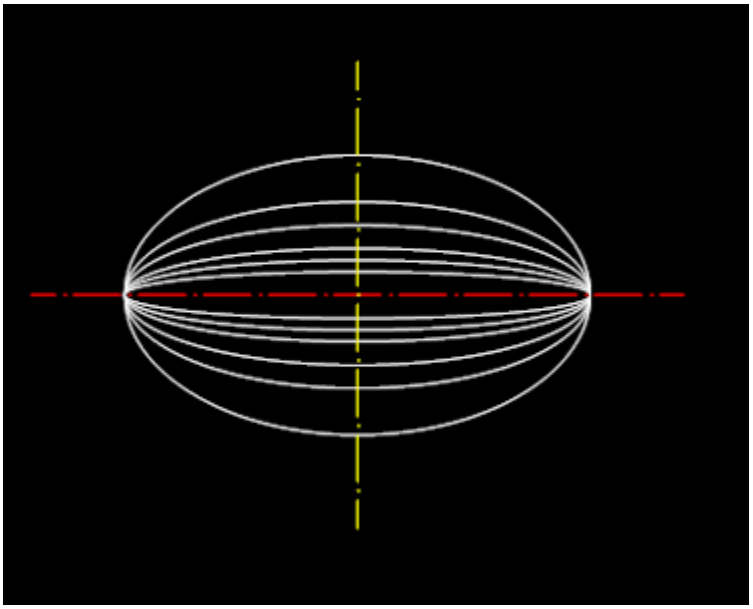
Parameters

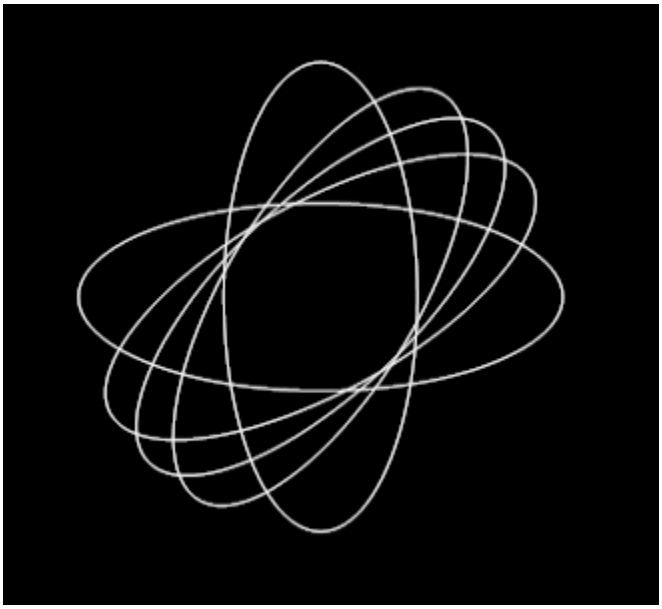
- **`center`** – center point (xy- or xyz-tuple), z-axis is 0 by default
- **`rx`** (*`float`*) – radius in x-axis
- **`ry`** (*`float`*) – radius in y-axis

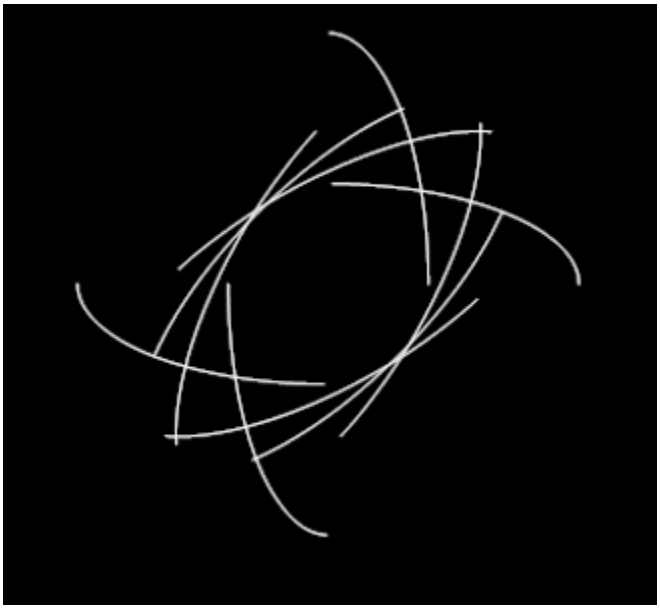
- **startangle** (*float*) – in degree
- **endangle** (*float*) – in degree
- **rotation** (*float*) – angle between x-axis and ellipse-main-axis in degree
- **segments** (*int*) – count of line segments for polyline approximation
- **linetype** (*str*) – linetype name, if not defined = *BYLAYER*
- **layer** (*str*) – layer name
- **color** (*int*) – range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*

Example

```
import dxfwrite
from dxfwrite import DXFEngine as dxf

name = 'ellipse.dxf'
dwg = dxf.drawing(name)

for axis in [0.5, 0.75, 1., 1.5,  2., 3.]:
    dwg.add(dxf.ellipse((0,0), 5., axis, segments=200))

dwg.add(dxf.line((-7, 0), (+7, 0), color=1, linetype='DASHDOT'))
dwg.add(dxf.line((0, -5), (0, +5), color=2, linetype='DASHDOT'))

for rotation in [0, 30, 45, 60, 90]:
    dwg.add(dxf.ellipse((20,0), 5., 2., rotation=rotation, segments=100))

for startangle in [0, 30, 45, 60, 90]:
    dwg.add(dxf.ellipse((40,0), 5., 2., startangle=startangle, endangle=startangle+90,
                        rotation=startangle, segments=90))
    dwg.add(dxf.ellipse((40,0), 5., 2., startangle=startangle+180,␣
↪endangle=startangle+270,
                        rotation=startangle, segments=90))

dwg.save()
```







Spline

Type: Composite Entity

Spline curves are approximated by *POLYLINE*.

For an explanation of spline curves see Wikipedia:

http://en.wikipedia.org/wiki/Spline_%28mathematics%29

DXFEngine.**spline**(*points*, *segments=100*, ***kwargs*)

Create a new cubic-spline-entity, curve shape is an approximation by *POLYLINE*.

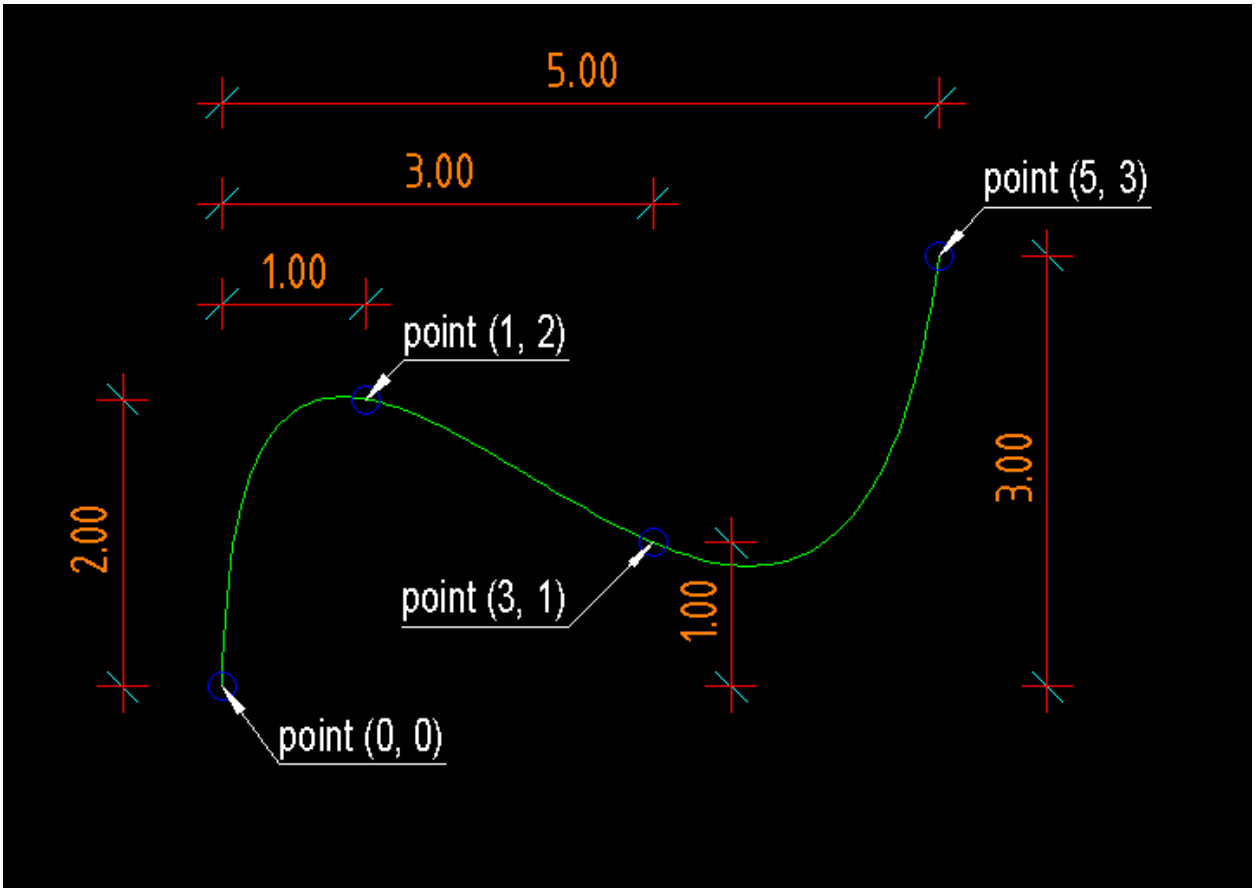
Parameters

- **points** – breakpoints (knots) as 2D points (float-tuples), defines the curve, the curve goes through this points
- **segments** (*int*) – count of line segments for polyline approximation
- **linetype** (*str*) – linetype name, if not defined = *BYLAYER*
- **layer** (*str*) – layer name
- **color** (*int*) – range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*

Example

```
import dxfwrite
from dxfwrite import DXFEngine as dxf

name = 'spline.dxf'
dwg = dxf.drawing(name)
#spline_points = [(0.0, 0.0), (1., 2.), (3., 1.), (5., 3.)]
spline_points = [(0.0, 0.0), (2., 2.), (3., 2.), (5., 0.)]
dwg.add(dxf.spline(spline_points, color=7))
```

```
for point in spline_points:
    dwg.add(dxf.circle(radius=0.1, center=point, color=1))
```



Bezier

Type: Composite Entity

Bezier curves are approximated by *POLYLINE*.

For an explanation of bezier curves see Wikipedia:

http://en.wikipedia.org/wiki/B%C3%A9zier_curve

bezier(*color=const.BYLAYER*, *layer='0'*, *linetype=None*)

Parameters

- **color** (*int*) – in range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*
- **layer** (*str*) – layer name
- **linetype** (*str*) – linetype name, if not defined = *BYLAYER*

Methods

Bezier.start(point, tangent):
Set start point and start tangent.

Parameters

- **point** – 2D start point
- **tangent** – start tangent as 2D vector, example: (5, 0) means a horizontal tangent with a length of 5 drawing units

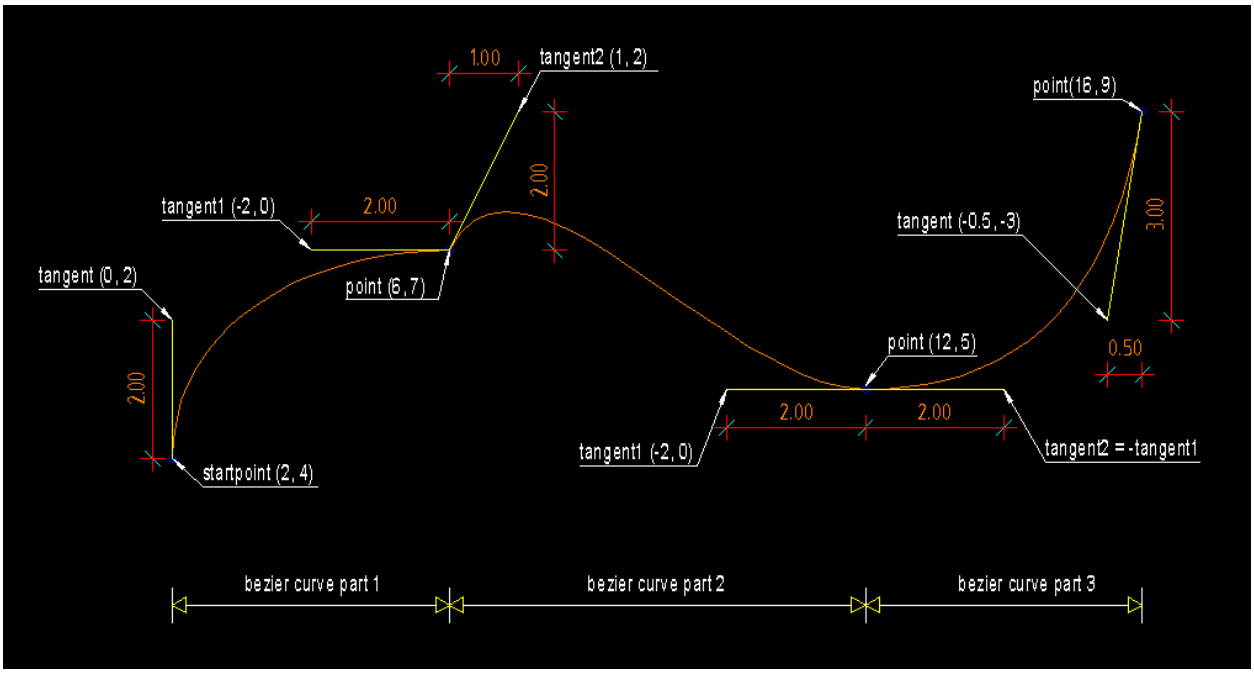
Bezier.**append**(*point*, *tangent1*, *tangent2=None*, *segments=20*)
Append a control point with two control tangents.

Parameters

- **point** – the control point as 2D point
- **tangent1** – first control tangent as 2D vector *left* of point
- **tangent2** – second control tangent as 2D vector *right* of point, if omitted tangent2 = -tangent1
- **segments** (*int*) – count of line segments for polyline approximation, count of line segments from previous control point to this point.

Example

```
import dxfwrite
from dxfwrite import DXFEngine as dxf
from dxfwrite.vector2d import vadd

def draw_control_point(point, tangent1, tangent2=(0, 0)):
    tp1 = vadd(point, tangent1)
    tp2 = vadd(point, tangent2)
    dwg.add(dxf.circle(0.05, center=point, color=1))
    dwg.add(dxf.line(point, tp1, color=2))
    dwg.add(dxf.line(point, tp2, color=2))

name = 'bezier.dxf'
dwg = dxf.drawing(name)
bezier = dxf.bezier(color=4)
dwg.add(bezier)

# define start point
bezier.start((2, 4), tangent=(0, 2))
draw_control_point((2, 4), (0, 2))

# append first point
bezier.append((6, 7), tangent1=(-2, 0), tangent2=(1, 2))
draw_control_point((6, 7), (-2, 0), (1, 2))

# tangent2 = -tangent1 = (+2, 0)
bezier.append((12, 5), tangent1=(-2, 0))
draw_control_point((12, 5), (-2, 0), (2, 0))

# for last point tangent2 is meaningless
bezier.append((16, 9), tangent1=(-0.5, -3))
draw_control_point((16, 9), (-0.5, -3))
```



Clothoid

Type: Composite Entity

Clothoid curves are approximated by *POLYLINE*.

For an explanation of clothoid curves see Wikipedia:

http://en.wikipedia.org/wiki/Clothoid

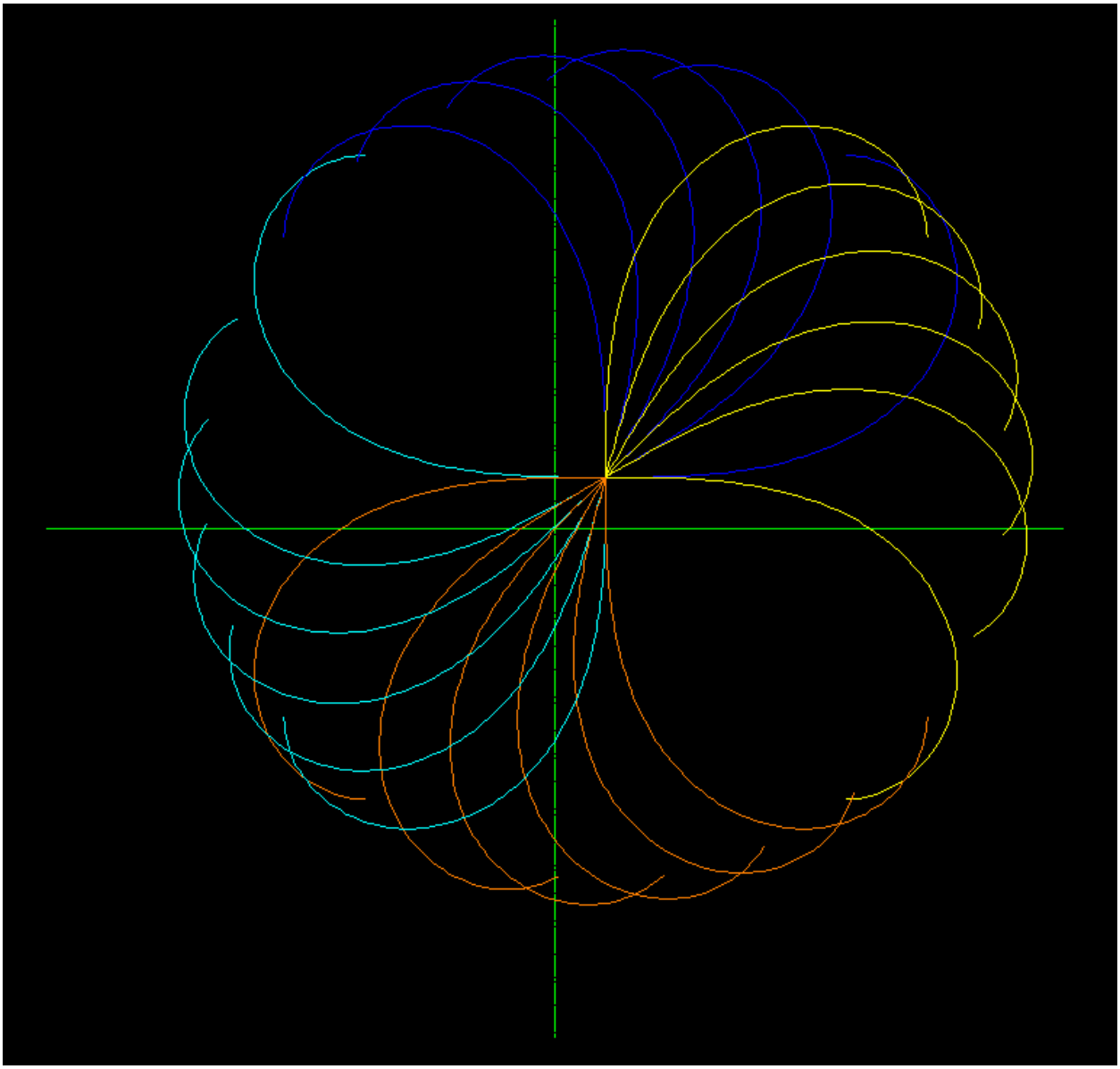
`DXFEngine.`**`clothoid`**(*start=(0, 0)*, *rotation=0.*, *length=1.*, *paramA=1.0*, *mirror=""*, *segments=100*, ***kwargs*)

Parameters

- **`start`** – insert point as 2D points (float-tuples)
- **`rotation`** (*`float`*) – in degrees
- **`length`** (*`loat`*) – length of curve in drawing units
- **`paramA`** (*`float`*) – clothoid parameter A
- **`mirror`** (*`str`*) – `'x'` for mirror curve about x-axis, `'y'` for mirror curve about y-axis, `'xy'` for mirror curve about x- and y-axis
- **`segments`** (*`int`*) – count of line segments for polyline approximation
- **`linetype`** (*`str`*) – linetype name, if not defined = *BYLAYER*
- **`layer`** (*`str`*) – layer name
- **`color`** (*`int`*) – range [1..255], 0 = *BYBLOCK*, 256 = *BYLAYER*

Example

```
import dxfwrite
from dxfwrite import DXFEngine as dxf

def four_c(A, length, rotation):
    dwg.add(dxf.clothoid(start=(2, 2), length=length, paramA=A,
                         rotation=rotation, color=1))
    dwg.add(dxf.clothoid(start=(2, 2), mirror='x', length=length, paramA=A,
                         rotation=rotation, color=2))
    dwg.add(dxf.clothoid(start=(2, 2), mirror='y', length=length, paramA=A,
                         rotation=rotation, color=3))
    dwg.add(dxf.clothoid(start=(2, 2), mirror='xy', length=length, paramA=A,
                         rotation=rotation, color=4))
name = 'clothoid.dxf'
dwg = dxf.drawing(name)
dwg.add(dxf.line((-20,0), (20, 0), linetype="DASHDOT2"))
dwg.add(dxf.line((0, -20), (0, 20), linetype="DASHDOT"))
for rotation in [0, 30, 45, 60, 75, 90]:
    four_c(10., 25, rotation)
```



CHAPTER 4

Howtos

Document Management

How to start using dxfwrite?

All DXF entities should be created by the *DXFEngine* object. To do that you have to import the DXF creation engine:

```
from dxfwrite import DXFEngine
```

How to create a new DXF drawing?

You can create a new DXF drawing by the *DXFEngine.drawing()* method:

```
drawing = DXFEngine.drawing('example.dxf')
```

How to manage global drawing settings?

The HEADER section of the DXF file contains settings of variables associated with the drawing. (for more informations see *HEADER Section*)

set/get header variables:

```
#set value
drawing.header['$ANGBASE'] = 30

#get value
version = drawing.header['$ACADVER'].value

# for 2D/3D points use:
minx, miny, minz = drawing.header['$EXTMIN'].tuple
```

How to create layers?

Layers are stored in the `layers` attribute in the *Drawing* class.

To create new layers just use:

```
drawing.new_layer('a new layer')
```

See also:

LAYER

Where are all the constants defined?

here:

```
from dxfwrite import const

drawing.new_layer('TEST', flags=const.LAYER_FROZEN)
```

How to create a new Textstyle?

Textstyles are stored in the `styles` attribute in the *Drawing* class.

To create a new Textstyle use:

```
drawing.new_style('BIGTEXT', height=12, font='arial.ttf')
```

See also:

TEXTSTYLE

How to insert XREFs?

AutoCAD will not always display the XREFs, other DXF-Viewers are less restrictive:

```
drawing.add_xref('path/drawing.dxf')
```

See also:

Drawing.add_xref()

Shapes Management

Prelude:

```
from dxfwrite import DXFEngine
dwg = DXFEngine.drawing('newdrawing.dxf')
```

How to create new Shapes?

Shapes like *LINE* or *CIRCLE* will be created by the *DXFEngine* object. A new created shape is not automatically added to the drawing, this is done by the *Drawing.add()* method of the *Drawing* object.

```
line = DXFEngine.line( (0, 0), (1, 1) )
dwg.add(line)
```

See also:

DXFEngine for available entities

How to set/get DXF attributes?

This is common to all **basic** DXF entities (not valid for composite entities):

```
# as keyword arguments
line = DXFEngine.line((0,0), (1,1), layer='TESTLAYER', linetype='DASHED', color=1)

# or:
line['layer'] = 'TESTLAYER'
line['linetype'] = 'DASHED'
line['color'] = 1
```

Where should the shapes be placed?

1. You can add the shapes to the drawing, which means adding the shape to the **model space**:

   ```
   line = DXFEngine.line((0, 0), (1, 1))
   dwg.add(line)
   ```

2. You can add the shape explicit to the **model space** of the drawing:

   ```
   dwg.modelspace.add(line)
   ```

3. You can add the shape to the **paper space** (layout) of the drawing:

   ```
   dwg.paperspace.add(line)
   ```

Note: The DXF R12 Standard supports only one paper space (layout).

4. You can add the shape to a **BLOCK** definition entity:

   ```
   blockdef = DXFEngine.block('testblk')
   blockdef.add(line)
   ```

Blocks Management

A block is a collection of objects grouped together to form a single object. All supported dxf entities can also be added to block like to a drawing.

Prelude:

```
from dxfwrite import DXFEngine
dwg = DXFEngine.drawing('newdrawing.dxf')
```

How to create a new block?

Just the blockname is required:

```
block = DXFEngine.block(name='rect')
```

How to add shapes to the block?

every supported shape can be added:

```
block.add(DXFEngine.rect((0, 0), width=3, height=3))
```

How use the block?

To use the block, insert the block by the *INSERT* entity:

```
dwg.add(DXFEngine.insert('rect', insert=(10,10)))
```

What is the basepoint?

The basepoint are the block coordinates, where the block will be placed by the insertion point of the *INSERT* entity.

How to use attributes?

Attributes are fill-in-the-blank text fields that you can add to your blocks. When you insert a block several times in a drawing, all the ordinary geometry (lines, circles, regular text strings, and so on) in all the instances are exactly identical. Attributes provide a little more flexibility in the form of text strings that can be different in each block insert.

```
# 1. create the ATTDEF
name = DXFEngine.attdef(tag='NAME', insert=(2, 2))
# 2. add the ATTDEF with the block
block.add(name)
```

Using INSERT

```
# 3. create a block-reference
blockref = DXFEngine.insert('rect', insert=(10,10))
# 4. create an ATTRIB, text is the ATTRIB content to display
nameattrib = name.new_attrib(text='Rect')
# 5. add ATTRIB to the block-reference
blockref.add(nameattrib)
# 6. add block-reference to dxf-drawing
dwg.add(blockref)
```

Using INSERT2

Simplified usage of attribs:

```
# 3. create a block-reference by insert2
# attribs = { key: value, ... }, key=ATTDEF-tag, value=ATTRIB-text
# but you have to use the block defintion
blockref = DXFEngine.insert2(block, insert=(10,10), attribs={'NAME': 'Rect'})
# 4. add block-reference to dxf-drawing
dwg.add(blockref)
```

Paper space (Layout)

In paper space you assemble your construction work done in model space to the form as you will print or plot it. In paper space you create *viewports* to the model space, you can also add headings, text, a drawing title block, a drawing frame and so on.

The paper space is separated from the model space by the common DXF entity attribute *paper_space*. If this attribute is `'0'` the entity is placed in model space, if the attribute is `'1'`, the entity is placed in paper space.

Note: In the DXF R12 format exists only **one** paper space.

How to add entities to paper space?

Use the *add()* method of the *Drawing* attribute *paperspace* to add entities to the paper space:

```
drawing.paperspace.add(line)
```

or just set the *paper_space* attribute of the DXF entity, and use the normal *add()* method of the *Drawing* object:

```
line = DXFEngine.line((0,0), (1,1), paper_space=1)
drawing.add(line)
```

Model space and paper space units

I always use the same units for model space and paper space, and because I am not an experienced AutoCAD user, I don't know if it is possible to choose different units in model space and paper space. For example: in model space 1 drawing unit = 1 meter, the paper space units are also in meters, so the paper space area of the DIN A0 paper format is 1.189 x 0.841 (meter/drawing units).

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

CHAPTER 6

Document License

Index

O

P

R

S

T

U

V