
ezdxf Documentation

Release 0.17.2

Manfred Moitzi

Jan 06, 2022

Contents

1 Included Extensions	3
2 Website	5
3 Documentation	7
4 Source Code & Feedback	9
5 Questions and Answers	11
6 Contents	13
6.1 Introduction	13
6.2 Setup & Dependencies	14
6.3 Usage for Beginners	21
6.4 Basic Concepts	25
6.5 Tutorials	41
6.6 Howto	185
6.7 FAQ	193
6.8 Reference	194
6.9 Launcher	511
6.10 Rendering	523
6.11 Add-ons	540
6.12 DXF Internals	590
6.13 Developer Guides	682
6.14 Glossary	699
6.15 Indices and tables	700
Python Module Index	701
Index	703

Welcome! This is the documentation for ezdxf release 0.17.2, last updated Jan 06, 2022.

- *ezdxf* is a Python package to create new DXF files and read/modify/write existing DXF files
- the intended audience are programmers
- requires at least Python 3.7
- OS independent
- additional required packages: [pyparsing](#)
- optional Cython implementation of some low level math classes
- MIT-License
- read/write/new support for DXF versions: R12, R2000, R2004, R2007, R2010, R2013 and R2018
- additional read support for DXF versions R13/R14 (upgraded to R2000)
- additional read support for older DXF versions than R12 (upgraded to R12)
- read/write support for ASCII DXF and Binary DXF
- preserves third-party DXF content

CHAPTER 1

Included Extensions

- `drawing` add-on to visualise and convert DXF files to images which can be saved to various formats such as png, pdf and svg
- `geo` add-on to support the `__geo_interface__`
- `r12writer` add-on to write basic DXF entities direct and fast into a DXF R12 file or stream
- `iterdxf` add-on to iterate over entities of the modelspace of really big (> 5GB) DXF files which do not fit into memory
- `importer` add-on to import entities, blocks and table entries from another DXF document
- `dxf2code` add-on to generate Python code for DXF structures loaded from DXF documents as starting point for parametric DXF entity creation
- `acadctb` add-on to read/write *Plot Style Files (CTB/STB)*
- `pycsg` add-on for Constructive Solid Geometry (CSG) modeling technique
- `MTextExplode` add-on for exploding MTEXT entities into single line TEXT entities

CHAPTER 2

Website

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

CHAPTER 3

Documentation

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

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

CHAPTER 4

Source Code & Feedback

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

Issue Tracker: <http://github.com/mozman/ezdxf/issues>

Forum: <https://github.com/mozman/ezdxf/discussions>

CHAPTER 5

Questions and Answers

Please post questions at the [forum](#) or [stack overflow](#) to make answers available to other users as well.

CHAPTER 6

Contents

6.1 Introduction

6.1.1 What is ezdxf

ezdxf is a [Python](#) interface to the [DXF](#) (drawing interchange file) format developed by Autodesk, *ezdxf* allows developers to read and modify existing DXF drawings or create new DXF drawings.

The main objective in the development of *ezdxf* was to hide complex DXF details from the programmer but still support most capabilities of the [DXF](#) format. Nevertheless, a basic understanding of the DXF format is required, also to understand which tasks and goals are possible to accomplish by using the the DXF format.

Not all DXF features are supported yet, but additional features will be added in the future gradually.

ezdxf is also a replacement for my [dxfwrite](#) and my [dxfgrabber](#) packages but with different APIs, for more information see also: [*What is the Relationship between ezdxf, dxfwrite and dxfgrabber?*](#)

6.1.2 What ezdxf can't do

- *ezdxf* is not a DXF converter: *ezdxf* can not convert between different DXF versions, if you are looking for an appropriate application, try the free [ODAFileConverter](#) from the [Open Design Alliance](#), which converts between different DXF version and also between the DXF and the DWG file format.
- *ezdxf* is not a CAD file format converter: *ezdxf* can not convert DXF files to other CAD formats such as DWG
- *ezdxf* is not a CAD kernel and does not provide high level functionality for construction work, it is just an interface to the DXF file format. If you are looking for a CAD kernel with [Python](#) scripting support, look at [FreeCAD](#).

6.1.3 Supported Python Versions

ezdxf requires at least Python 3.7 and will be tested with the latest stable CPython version and the latest stable release of pypy3 during development.

ezdxf is written in pure Python with optional Cython implementations of some low level math classes and requires only *pyparser* and *typing_extensions* as additional library beside the Python Standard Library. *pytest* is required to run the unit and integration tests. Data to run the stress and audit test can not be provided, because I don't have the rights for publishing this DXF files.

6.1.4 Supported Operating Systems

ezdxf is OS independent and runs on all platforms which provide an appropriate Python interpreter (≥ 3.7).

6.1.5 Supported DXF Versions

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

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

6.1.6 Embedded DXF Information of 3rd Party Applications

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

6.1.7 License

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

6.2 Setup & Dependencies

The primary goal is to keep the dependencies of the *core* package as small as possible. The add-ons are not part of the core package and can therefore use as many packages as needed. The only requirement for these packages is an easy way to install them on *Windows*, *Linux* and *macOS*, preferably as:

```
pip3 install eздxf
```

The *pyparsing* package and the *typing_extensions* are the only hard dependency and will be installed automatically by *pip3*!

The minimal required Python version is determined by the latest stable version of *pypy3* and the Python version deployed by the *Raspberry Pi* OS, which is currently Python 3.7 (2021).

6.2.1 Basic Installation

The most common case is the installation by `pip3` including the optional C-extensions from PyPI as binary wheels:

```
pip3 install eздxf
```

6.2.2 Installation with Extras

To use all features of the drawing add-on, add the `[draw]` tag:

```
pip3 install eздxf[draw]
```

Tag	Additional Installed Packages
<code>[draw]</code>	<code>Matplotlib</code> , <code>PySide6</code>
<code>[draw5]</code>	<code>Matplotlib</code> , <code>PyQt5</code> (use only if <code>PySide6</code> is not available)
<code>[test]</code>	<code>geomdl</code> , <code>pytest</code>
<code>[dev]</code>	<code>setuptools</code> , <code>wheel</code> , <code>Cython</code> + <code>[test]</code>
<code>[all]</code>	<code>[draw]</code> + <code>[test]</code> + <code>[dev]</code>
<code>[all5]</code>	<code>[draw5]</code> + <code>[test]</code> + <code>[dev]</code> (use only if <code>PySide6</code> is not available)

6.2.3 Binary Wheels

Ezdx includes some C-extensions, which will be deployed automatically at each release to PyPI as binary wheels to PyPI:

- *Windows*: only amd64 packages
- *Linux*: manylinux and musllinux packages for x86_64 & aarch64
- *macOS*: x86_64, arm64 and universal packages

The wheels are created by the continuous integration (CI) service provided by GitHub and the build container cibuildwheel provided by PyPA the Python Packaging Authority. The workflows are kept short and simple, so my future me will understand what's going on and they are maybe also helpful for other developers which do not touch CI services every day.

The C-extensions are disabled for pypy3, because the JIT compiled code of pypy is much faster than the compiled C-extensions for pypy.

6.2.4 Disable C-Extensions

It is possible to disable the C-Extensions by setting the environment variable `EZDXF_DISABLE_C_EXT` to 1 or true:

```
set EZDXF_DISABLE_C_EXT=1
```

or on Linux:

```
export EZDXF_DISABLE_C_EXT=1
```

This has to be done **before** anything from ezdx is imported! If you are working in an interactive environment, you have to restart the interpreter.

6.2.5 Installation from GitHub

Install the latest development version by *pip3* from [GitHub](#):

```
pip3 install git+https://github.com/mozman/ezdxf.git@master
```

6.2.6 Build and Install from Source

This is only required if you want the compiled C-extensions, the *ezdxf* installation by *pip* from the source code package works without the C-extension but is slower. There are many binary wheels including the compiles C-extensions available on [PyPi](#).

Windows 10

Make a build directory and a virtual environment:

```
mkdir build
cd build
py -m venv py310
py310/Scripts/activate.bat
```

A working C++ compiler setup is required to compile the C-extensions from source code. Windows users need the build tools from Microsoft: <https://visualstudio.microsoft.com/de/downloads/>

Download and install the required Visual Studio Installer of the community edition and choose the option: *Visual Studio Build Tools 20..*

Install required packages to build and install *ezdxf* with C-extensions:

```
pip3 install setuptools wheel cython
```

Clone the [GitHub](#) repository:

```
git clone https://github.com/mozman/ezdxf.git
```

Build and install *ezdxf* from source code:

```
cd eздxf
pip3 install .
```

Check if the installation was successful:

```
python3 -m eздxf -V
```

The *ezdxf* command should run without a preceding *python3 -m*, but calling the launcher through the interpreter guarantees to call the version which was installed in the venv if there exist a global installation of *ezdxf* like in my case.

The output should look like this:

```
ezdxf 0.17.2b4 from D:\Source\build\py310\lib\site-packages\ezdxf
Python version: 3.10.1 (tags/v3.10.1:2cd268a, Dec 6 2021, 19:10:37) [MSC v.1929 64_
bit (AMD64)]
using C-extensions: yes
using Matplotlib: no
```

To install optional packages go to section: [Install Optional Packages](#)

To run the included tests go to section: [Run the Tests](#)

WSL & Ubuntu

I use sometimes the Windows Subsystem for Linux ([WSL](#)) with [Ubuntu](#) 20.04 LTS for some tests (how to install [WSL](#)).

By doing as fresh install on *WSL & Ubuntu*, I encountered an additional requirement, the *build-essential* package adds the required C++ support:

```
sudo apt install build-essential
```

The system Python 3 interpreter has the version 3.8, but I will show in a later section how to install an additional newer Python version from the source code:

```
cd ~
mkdir build
cd build
python3 -m venv py38
source py38/bin/activate
```

Install *Cython* and *wheel* in the venv to get the C-extensions compiled:

```
pip3 install cython wheel
```

Clone the [GitHub](#) repository:

```
git clone https://github.com/mozman/ezdxf.git
```

Build and install ezdxf from source code:

```
cd ezdxf
pip3 install .
```

Check if the installation was successful:

```
python3 -m ezdxf -V
```

The output should look like this:

```
ezdxf 0.17.2b4 from /home/mozman/src/py38/lib/python3.8/site-packages/ezdxf
Python version: 3.8.10 (default, Nov 26 2021, 20:14:08)
[GCC 9.3.0]
using C-extensions: yes
using Matplotlib: no
```

To install optional packages go to section: [Install Optional Packages](#)

To run the included tests go to section: [Run the Tests](#)

Raspberry Pi OS

Testing platform is a [Raspberry Pi 400](#) and the OS is the [Raspberry Pi OS](#) which runs on 64bit hardware but is a 32bit OS. The system Python 3 interpreter comes in version 3.7, but I will show in a later section how to install an additional newer Python version from the source code.

Install the build requirements, [Matplotlib](#) and the [PyQt5](#) bindings from the distribution repository:

```
sudo apt install python3-pip python3-matplotlib python3-pyqt5
```

Installing [Matplotlib](#) and the [PyQt5](#) bindings by *pip* from [piwheels](#) in the venv worked, but the packages showed errors at import, seems to be an packaging error in the required [numpy](#) package. [PySide6](#) is the preferred Qt binding but wasn't available on [Raspberry Pi OS](#) at the time of writing this - [PyQt5](#) is supported as fallback.

Create the venv with access to the system site-packages for using [Matplotlib](#) and the Qt bindings from the system installation:

```
cd ~  
mkdir build  
cd build  
python3 -m venv --system-site-packages py37  
source py37/bin/activate
```

Install [Cython](#) and [wheel](#) in the venv to get the C-extensions compiled:

```
pip3 install cython wheel
```

Clone the [GitHub](#) repository:

```
git clone https://github.com/mozman/ezdxf.git
```

Build and install ezdxf from source code:

```
cd eздxf  
pip3 install .
```

Check if the installation was successful:

```
python3 -m eздxf -V
```

The output should look like this:

```
ezdxf 0.17.2b4 from /home/pi/src/py37/lib/python3.7/site-packages/ezdxf  
Python version: 3.7.3 (default, Jan 22 2021, 20:04:44)  
[GCC 8.3.0]  
using C-extensions: yes  
using Matplotlib: yes
```

To run the included tests go to section: [Run the Tests](#)

Manjaro on Raspberry Pi

Because the (very well working) [Raspberry Pi OS](#) is only a 32bit OS, I searched for a 64bit alternative like [Ubuntu](#), which just switched to version 21.10 and always freezes at the installation process! So I tried [Manjaro](#) as rolling release, which I used prior in a virtual machine and wasn't really happy, because there is always something to update. Anyway the distribution looks really nice and has Python 3.9.9 installed.

Install build requirements and optional packages by the system packager *pacman*:

```
sudo pacman -S python-pip python-matplotlib python-pyqt5
```

Create and activate the venv:

```
cd ~
mkdir build
cd build
python3 -m venv --system-site-packages py39
source py39/bin/activate
```

The rest is the same procedure as for the [Raspberry Pi OS](#):

```
pip3 install cython wheel
git clone https://github.com/mozman/ezdxf.git
cd eздxf
pip3 install .
python3 -m eздxf -V
```

To run the included tests go to section: [Run the Tests](#)

Ubuntu Server 21.10 on Raspberry Pi

I gave the [Ubuntu](#) Server 21.10 a chance after the desktop version failed to install by a nasty bug and it worked well. The distribution comes with Python 3.9.4 and after installing some requirements:

```
sudo apt install build-essential python3-pip python3.9-venv
```

The remaining process is like on [WSL & Ubuntu](#) except for the newer Python version. Installing [Matplotlib](#) by [pip](#) works as expected and is maybe useful even on a headless server OS to create SVG and PNG from DXF files. [PySide6](#) is not available by [pip](#) and the installation of [PyQt5](#) starts from the source code package which I stopped because this already didn't finished on [Manjaro](#), but the installation of the [PyQt5](#) bindings by [apt](#) works:

```
sudo apt install python3-pyqt5
```

Use the `--system-site-packages` option for creating the venv to get access to the [PyQt5](#) package.

6.2.7 Install Optional Packages

Install the optional dependencies by [pip](#) only for [Windows 10](#) and [WSL & Ubuntu](#), for [Raspberry Pi OS](#) and [Manjaro on Raspberry Pi](#) install these packages by the system packager:

```
pip3 install matplotlib PySide6
```

6.2.8 Run the Tests

This is the same procedure for all systems, assuming you are still in the build directory `build/ezdxf` and `ezdxf` is now installed in the venv.

Install the test dependencies and run the tests:

```
pip3 install pytest geomdl
python3 -m pytest tests integration_tests
```

6.2.9 Build Documentation

Assuming you are still in the build directory *build/ezdxf* of the previous section.

Install Sphinx:

```
pip3 install Sphinx sphinx-rtd-theme
```

Build the HTML documentation:

```
cd docs  
make html
```

The output is located in *build/ezdxf/docs/build/html*.

6.2.10 Python from Source

Debian based systems have often very outdated software installed and sometimes there is no easy way to install a newer Python version. This is a brief summery how I installed Python 3.9.9 on the [Raspberry Pi OS](#), for more information go to the source of the recipe: [Real Python](#)

Install build requirements:

```
sudo apt-get update  
sudo apt-get upgrade  
  
sudo apt-get install -y make build-essential libssl-dev zlib1g-dev \  
libbz2-dev libreadline-dev libsqlite3-dev wget curl llvm \  
libncurses5-dev libncursesw5-dev xz-utils tk-dev
```

Make a build directory:

```
cd ~  
mkdir build  
cd build
```

Download and unpack the source code from [Python.org](#), replace 3.9.9 by your desired version:

```
wget https://www.python.org/ftp/python/3.9.9/Python-3.9.9.tgz  
tar -xvzf Python-3.9.9.tgz  
cd Python-3.9.9/
```

Configure the build process, use a prefix to the directory where the interpreter should be installed:

```
./configure --prefix=/opt/python3.9.9 --enable-optimizations
```

Build & install the Python interpreter. The *-j* option simply tells *make* to split the building into parallel steps to speed up the compilation, my [Raspberry Pi](#) 400 has 4 cores so 4 seems to be a good choice:

```
make -j 4  
sudo make install
```

The building time was ~25min and the new Python 3.9.9 interpreter is now installed as */opt/python3.9.9/bin/python3*.

At the time there were no system packages for [Matplotlib](#) and [PyQt5](#) for this new Python version available, so there is no benefit of using the option *--system-site-packages* for building the venv:

```
cd ~/build
/opt/python3.9.9/bin/python3 -m venv py39
source py39/bin/activate
```

I have not tried to build Matplotlib and PyQt5 by myself and the installation by *pip* from [piwheels](#) did not work, in this case you don't get Matplotlib support for better font measuring and the *drawing* add-on will not work.

Proceed with the *ezdxf* installation from source as shown for the *Raspberry Pi OS*.

6.3 Usage for Beginners

This section shows the intended usage of the *ezdxf* package. This is just a brief overview for new *ezdxf* users, follow the provided links for more detailed information.

First import the package:

```
import eздxf
```

6.3.1 Loading DXF Files

ezdxf supports loading ASCII and binary DXF files from a file:

```
doc = eздxf.readfile(filename)
```

or from a zip-file:

```
doc = e Zukxf.readzip(zipfilename[, filename])
```

Which loads the DXF file *filename* from the zip-file *zipfilename* or the first DXF file in the zip-file if *filename* is absent.

It is also possible to read a DXF file from a stream by the *e Zukxf.read()* function, but this is a more advanced feature, because this requires detection of the file encoding in advance.

This works well with DXF files from trusted sources like AutoCAD or BricsCAD, for loading DXF files with minor or major flaws look at the *e Zukxf.recover* module.

See also:

Documentation for *e Zukxf.readfile()*, *e Zukxf.readzip()* and *e Zukxf.read()*, for more information about file management go to the *Document Management* section. For loading DXF files with structural errors look at the *e Zukxf.recover* module.

6.3.2 Saving DXF Files

Save the DXF document with a new name:

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

or with the same name as loaded:

```
doc.save()
```

See also:

Documentation for `ezdxf.document.Drawing.save()` and `ezdxf.document.Drawing.saveas()`, for more information about file management go to the [Document Management](#) section.

6.3.3 Create a New DXF File

Create new file for the latest supported DXF version:

```
doc = ezdxf.new()
```

Create a new DXF file for a specific DXF version, e.g for DXF R12:

```
doc = ezdxf.new("R12")
```

To setup some basic DXF resources use the `setup` argument:

```
doc = ezdxf.new(setup=True)
```

See also:

Documentation for `ezdxf.new()`, for more information about file management go to the [Document Management](#) section.

6.3.4 Layouts and Blocks

Layouts are containers for DXF entities like LINE or CIRCLE. The most important layout is the modelspace labeled as “Model” in CAD applications which represents the “world” work space. Paperspace layouts represents plottable sheets which contains often the framing and the tile block of a drawing and VIEWPORT entities as scaled and clipped “windows” into the modelspace.

The modelspace is always present and can not be deleted. The active paperspace is also always present in a new DXF document but can be deleted, in that case another paperspace layout gets the new active paperspace, but you can not delete the last paperspace layout.

Getting the modelspace of a DXF document:

```
msp = doc.modelspace()
```

Getting a paperspace layout by the name as shown in the tab of a CAD application:

```
psp = doc.layout("Layout1")
```

A block is just another kind of entity space, which can be inserted multiple times into other layouts and blocks by the INSERT entity also called block references, this is a very powerful and important concept of the DXF format.

Getting a block layout by the block name:

```
blk = doc.blocks.get("NAME")
```

All these layouts have factory functions to create graphical DXF entities for their entity space, for more information about creating entities see section: [Create new DXF Entities](#)

6.3.5 Create New Blocks

The block definitions of a DXF document are managed by the `BlocksSection` object:

```
my_block = doc.blocks.new("MyBlock")
```

See also:

[Tutorial for Blocks](#)

6.3.6 Query DXF Entities

As said in the [Layouts and Blocks](#) section, all graphical DXF entities are stored in layouts, all these layouts can be iterated and support the index operator e.g. `layout[-1]` returns the last entity.

The main difference between iteration and index access is, that iteration filters destroyed entities, but the index operator returns also destroyed entities until these entities are purged by `layout.purge()` more about this topic in section: [Delete Entities](#).

There are two advanced query methods: `query()` and `groupby()`.

Get all lines of layer "MyLayer":

```
lines = msp.query('LINE[layer=="MyLayer"]')
```

This returns an `EntityQuery` container, which also provides the same `query()` and `groupby()` methods.

Get all lines categorized by a DXF attribute like color:

```
all_lines_by_color = msp.query("LINE").groupby("color")
lines_with_color_1 = all_lines_by_color.get(1, [])
```

The `groupby()` method returns a regular Python `dict` with colors as key and a regular Python `list` of entities as values (not an `EntityQuery` container).

See also:

For more information go to the [Tutorial for getting data from DXF files](#)

6.3.7 Examine DXF Entities

Each DXF entity has a `dxf` namespace attribute, which stores the named DXF attributes, some DXF attributes are only indirect available like the vertices in the LWPOLYLINE entity. More information about the DXF attributes of each entity can found in the documentation of the `ezdxf.entities` module.

Get some basic DXF attributes:

```
layer = entity.dxf.layer # default is "0"
color = entity.dxf.color # default is 256 = BYLAYER
```

Most DXF attributes have a default value, which will be returned if the DXF attribute is not present, for DXF attributes without a default value you can check in the attribute really exist:

```
entity.dxf.hasattr("true_color")
```

or use the `get()` method and a default value:

```
entity.dxf.get("true_color", 0)
```

See also:

Common graphical DXF attributes

6.3.8 Create New DXF Entities

The factory methods for creating new graphical DXF entities are located in the `BaseLayout` class. This means this factory methods are available for all entity containers:

- `Modelspace`
- `Paperspace`
- `BlockLayout`

The usage is simple:

```
msp = doc.modelspace()
msp.add_line((0, 0), (1, 0), dxfattribs={"layer": "MyLayer"})
```

See also:

Thematic Index of Layout Factory Methods

A few important or required DXF attributes are explicit method arguments, most additional and optional DXF attributes are given as a regular Python dict object. The supported DXF attributes can be found in the documentation of the `ezdxf.entities` module.

Warning: Do not instantiate DXF entities by yourself and add them to layouts, always use the provided factory function to create new graphical entities, this is the intended way to use `ezdxf`.

6.3.9 Create Block References

A block reference is just another DXF entity called INSERT, but the term “Block Reference” is a better choice and so the `Insert` entity is created by the factory function: `add_blockref()`:

```
msp.add_blockref("MyBlock", (0, 0))
```

See also:

See *Tutorial for Blocks* for more advanced features like using `Attrib` entities.

6.3.10 Create New Layers

A layer is not an entity container, a layer is just another DXF attribute stored in the entity and this entity can inherit some properties from this `Layer` object. Layer objects are stored in the layer table which is available as attribute `doc.layers`.

You can create your own layers:

```
my_layer = doc.layer.add("MyLayer")
```

The layer object also controls the visibility of entities which references this layer, the on/off state of the layer is unfortunately stored as positive or negative color value which make the raw DXF attribute of layers useless, to change the color of a layer use the property `Layer.color`

```
my_layer.color = 1
```

To change the state of a layer use the provided methods of the `Layer` object, like `on()`, `off()`, `freeze()` or `thaw()`:

```
my_layer.off()
```

See also:

[Layer Concept](#)

6.3.11 Delete Entities

The safest way to delete entities is to delete the entity from the layout containing that entity:

```
line = msp.add_line((0, 0), (1, 0))
msp.delete_entity(line)
```

This removes the entity immediately from the layout and destroys the entity. The property `is_alive` returns `False` for a destroyed entity and all Python attributes are deleted, so `line.dxf.color` will raise an `AttributeError` exception, because `line` does not have a `dxf` attribute anymore.

`Ezdx`f also supports also destruction of entities by calling method `destroy()` manually:

```
line.destroy()
```

Manually destroyed entities are not removed immediately from entities containers like `Modelspace` or `EntityQuery`, but iterating such a container will filter destroyed entities automatically, so a `for e in msp: ...` loop will never yield destroyed entities. The index operator and the `len()` function do **not** filter deleted entities, to avoid getting deleted entities call the `purge()` method of the container manually to remove deleted entities.

6.3.12 Further Information

- [Reference](#) documentation
- Documentation of package internals: [Developer Guides](#).

6.4 Basic Concepts

The Basic Concepts section teach the intended meaning of DXF attributes and structures without teaching the application of this information or the specific implementation by `ezdxf`, if you are looking for more information about the `ezdxf` internals look at the [Reference](#) section or if you want to learn how to use `ezdxf` go to the [Tutorials](#) section and for the solution of specific problems go to the [Howto](#) section.

6.4.1 What is DXF?

The common assumption is also the cite of [Wikipedia](#):

AutoCAD DXF (Drawing eXchange Format) is a CAD data file format developed by Autodesk for enabling data interoperability between AutoCAD and **other** applications.

DXF was originally introduced in December 1982 as part of AutoCAD 1.0, and was intended to provide an exact representation of the data in the AutoCAD native file format, DWG (Drawing). For many years Autodesk did not publish specifications making correct imports of DXF files difficult. Autodesk now publishes the DXF specifications online.

The more precise cite from the [DXF reference](#) itself:

The DXF™ format is a tagged data representation of all the information contained in an AutoCAD® drawing file. Tagged data means that each data element in the file is preceded by an integer number that is called a group code. A group code's value indicates what type of data element follows. This value also indicates the meaning of a data element for a given object (or record) type. Virtually all user-specified information in a drawing file can be represented in DXF format.

No mention of interoperability between AutoCAD and **other** applications.

In reality the DXF format was designed to ensure AutoCAD cross-platform compatibility in the early days when different hardware platforms with different binary data formats were used. The name DXF (Drawing eXchange Format) may suggest an universal exchange format, but it is not. It is based on the infrastructure installed by Autodesk products (fonts) and the implementation details of AutoCAD (MTEXT) or on licensed third party technologies (embedded ACIS entities).

For more information about the AutoCAD history see the document: [The Autodesk File - Bits of History, Words of Experience](#) by *John Walker*, founder of *Autodesk, Inc.* and co-author of *AutoCAD*.

DXF Reference Quality

The [DXF reference](#) is by far no specification nor a standard like the W3C standard for [SVG](#) or the ISO standard for [PDF](#).

The reference describes many but not all DXF entities and some basic concepts like the tag structure or the arbitrary axis algorithm. But the existing documentation (reference) is incomplete and partly misleading or wrong. Also missing from the reference are some important parts like the complex relationship between the entities to create higher order structures like block definitions, layouts (model space & paper space) or dynamic blocks to name a few.

Reliable CAD Applications

Because of the suboptimal quality of the DXF reference not all DXF viewers, creators or processors are of equal quality. I consider a CAD application as a *reliable CAD application* when the application creates valid DXF documents in the meaning and interpretation of [Autodesk](#) and a reliable DXF viewer when the result matches in most parts the result of the free [Trueview](#) viewer provided by Autodesk.

These are some application which do fit the criteria of a reliable CAD application:

- [AutoCAD](#) and [Trueview](#)
- CAD applications based on the [OpenDesignAlliance](#) (ODA) SDK, see also [ODA on wikipedia](#), even [Autodesk](#) is a corporate member, see their blog post from [22 Sep 2020](#) at [adsknews](#) but only to use the ODA IFC tools and not to improve the DWG/DXF compatibility
- [BricsCAD](#) (ODA based)
- [GstarCAD](#) (ODA based)
- [ZWCAD](#) (ODA based)

Unfortunately, I cannot recommend any open source applications because everyone I know has serious shortcomings, at least as a DXF viewer, and I don't trust them as a DXF creator either. To be clear, even *ezdxf* (which is not a CAD application) is a *reliable* library in this sense - it just keeps getting better, but is far from *reliable*.

BTW: Don't send bug reports based on [LibreCAD](#) or [QCAD](#), I won't waste my time on them.

6.4.2 AutoCAD Color Index (ACI)

The `color` attribute represents an *ACI* (AutoCAD Color Index). AutoCAD and many other *CAD* application provides a default color table, but pen table would be the more correct term. Each ACI entry defines the color value, the line weight and some other attributes to use for the pen. This pen table can be edited by the user or loaded from an *CTB* or *STB* file. *Ezdx*f provides functions to create (`new()`) or modify (`ezdxf.acadctb.load()`) plot styles files.

DXF R12 and prior are not good in preserving the layout of a drawing, because of the lack of a standard color table defined by the DXF reference and missing DXF structures to define these color tables in the DXF file. So if a CAD user redefined an ACI and do not provide a *CTB* or *STB* file, you have no ability to determine which color or linewidth was used. This is better in later DXF versions by providing additional DXF attributes like `lineweight` and `true_color`.

See also:

[Plot Style Files \(CTB/STB\) `ezdxf.colors`](#)

6.4.3 Layer Concept

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

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

Create a layer table entry `Layer` by `Drawing.layers.new()`, assign the layer properties such as color and linetype. Then assign those layers to other DXF entities by setting the DXF attribute `layer` to the layer name as string.

It is possible to use layers without a layer definition but not recommend, just use a layer name without a layer definition, the layer has the default linetype 'Continuous' and the default color is 7.

The advantage of assigning a linetype and a color to a layer is that entities on this layer can inherit this properties by using 'BYLAYER' as linetype string and 256 as color, both values are default values for new entities.

See also:

[Tutorial for Layers](#)

6.4.4 Linetypes

The `linetype` defines the pattern of a line. The linetype of an entity can be specified by the DXF attribute `linetype`, this can be an explicit named linetype or the entity can inherit its line type from the assigned layer by setting `linetype` to 'BYLAYER', which is also the default value. CONTINUOUS is the default line type for layers with unspecified line type.

ezdxf creates several standard linetypes, if the argument *setup* is `True` at calling `new()`, this simple line types are supported by all DXF versions:

```
doc = ezdxf.new('R2007', setup=True)
```

CONTINUOUS



CENTER



CENTERX2



CENTER2



DASHED



DASHEDX2



DASHED2



PHANTOM



PHANTOMX2



PHANTOM2



DASHDOT



DASHDOTX2



DASHDOT2



DOT



DOTX2



DOT2



DIVIDE



In DXF R13 Autodesk introduced complex linetypes, containing TEXT or SHAPES in linetypes. *ezdxf* v0.8.4 and later supports complex linetypes.

See also:

Tutorial for Linetypes

Linetype Scaling

Global linetype scaling can be changed by setting the header variable `doc.header['$LTSCALE'] = 2`, which stretches the line pattern by factor 2.

To change the linetype scaling for single entities set scaling factor by DXF attribute `ltscale`, which is supported since DXF version R2000.

6.4.5 Lineweights

The `lineweight` attribute represents the linewidth as integer value in millimeters * 100, e.g. 0.25mm = 25, independently from the unit system used in the DXF document. The `lineweight` attribute is supported by DXF version R2000 and newer.

Only certain values are valid, they are stored in `ezdxf.lldxf.const.VALID_DXF_LINEWEIGHTS`: 0, 5, 9, 13, 15, 18, 20, 25, 30, 35, 40, 50, 53, 60, 70, 80, 90, 100, 106, 120, 140, 158, 200, 211.

Values < 0 have a special meaning and can be imported as constants from `ezdxf.lldxf.const`

-1	LINEWEIGHT_BYLAYER
-2	LINEWEIGHT_BYBLOCK
-3	LINEWEIGHT_DEFAULT

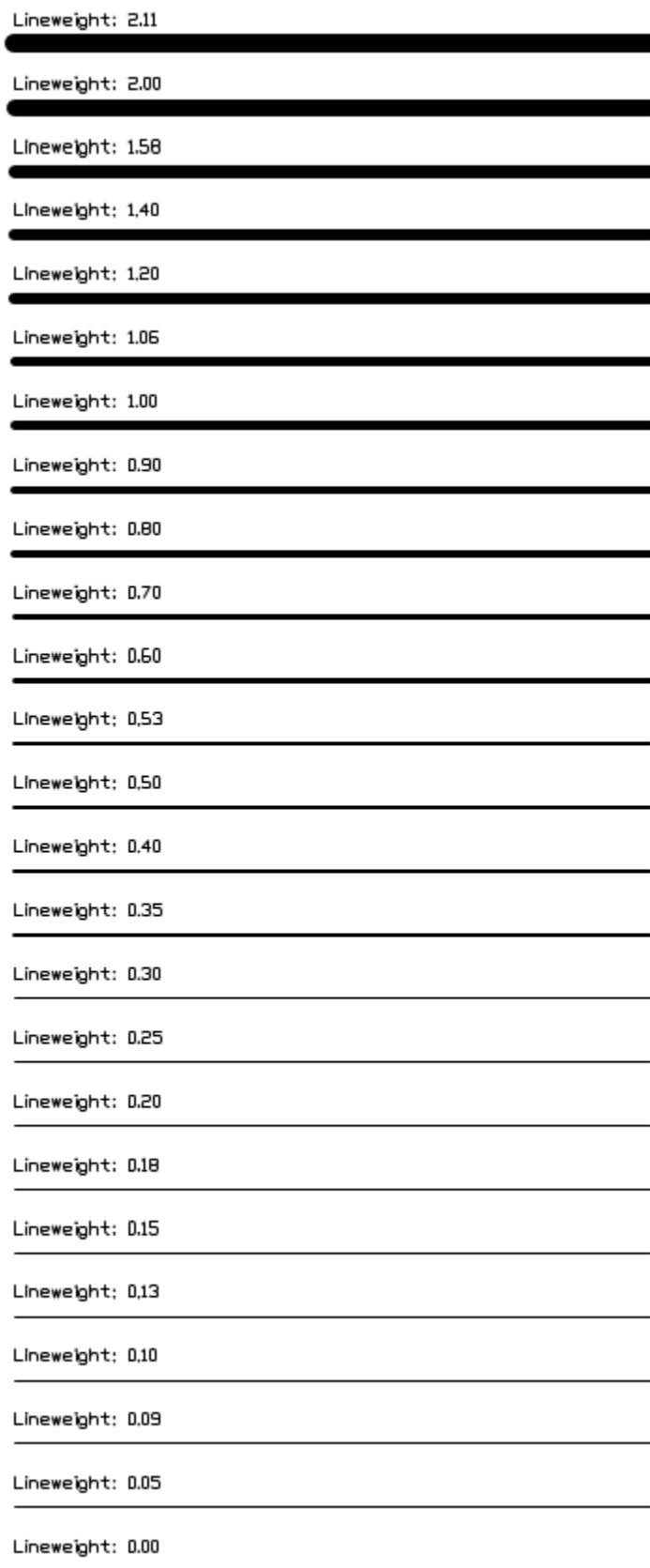
The validator function: `ezdxf.lldxf.validator.is_valid_lineweight()` returns True for valid linewidth values otherwise False.

Sample file which shows all valid lineweights: `valid_lineweights.dxf`

You have to enable the option to show lineweights in your CAD application or viewer to see the effect on screen, which is often disabled by default, the same has to be done in the page setup options for plotting lineweights.

Setting the HEADER variable `$LWDISPLAY` to 1 may activate support for showing lineweights on screen and `$LWDISPSCALE` may scale the linewidth on screen:

```
# activate on screen linewidth display
doc.header["$LWDISPLAY"] = 1
# linewidth scaling factor for on screen display
doc.header["$LWDISPSCALE"] = 0.55
```



The linewidth value can be overridden by [CTB](#) or [STB](#) files.

6.4.6 Coordinate Systems

[AutoLISP Reference to Coordinate Systems](#) provided by Autodesk.

To brush up your knowledge about vectors, watch the YouTube tutorials of [3Blue1Brown](#) about [Linear Algebra](#).

WCS

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

UCS

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

User defined coordinate systems are not just helpful for interactive CAD, therefore ezdxf provides a converter class [UCS](#) to translate coordinates from UCS into WCS and vice versa, but always remember: store only WCS or OCS coordinates in DXF files, because there is no method to determine which UCS was active or used to create UCS coordinates.

See also:

- Table entry [UCS](#)
- [ezdxf.math.UCS](#) - converter between WCS and UCS

OCS

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

Because *ezdxf* is just an interface to DXF, it does not automatically convert OCS into WCS, this is the domain of the user/application. And further more, the main goal of OCS is to place 2D elements in 3D space, this maybe was useful in the early years of CAD, I think nowadays this is an not often used feature, but I am not an AutoCAD user.

OCS differ from WCS only if extrusion != (0, 0, 1), convert OCS into WCS:

```
# circle is an DXF entity with extrusion != (0, 0, 1)
ocs = circle.ocs()
wcs_center = ocs.to_wcs(circle.dxf.center)
```

See also:

- [Object Coordinate System \(OCS\)](#) - deeper insights into OCS
- [ezdxf.math.OCS](#) - converter between WCS and OCS

DCS

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

6.4.7 Object Coordinate System (OCS)

- DXF Reference for OCS provided by Autodesk.

The points associated with each entity are expressed in terms of the entity's own object coordinate system (OCS). The OCS was referred to as ECS in previous releases of AutoCAD.

With OCS, the only additional information needed to describe the entity's position in 3D space is the 3D vector describing the z-axis of the OCS (often referenced as extrusion vector), and the elevation value, which is the distance of the entity xy-plane to the WCS/OCS origin.

For a given z-axis (extrusion) direction, there are an infinite number of coordinate systems, defined by translating the origin in 3D space and by rotating the x- and y-axis around the z-axis. However, for the same z-axis direction, there is only one OCS. It has the following properties:

- Its origin coincides with the WCS origin.
- The orientation of the x- and y-axis within the xy-plane are calculated in an arbitrary but consistent manner. AutoCAD performs this calculation using the arbitrary axis algorithm (see below).
- Because of *Arbitrary Axis Algorithm* the OCS can only represent a **right-handed** coordinate system!

The following entities do not lie in a particular plane. All points are expressed in world coordinates. Of these entities, only lines and points can be extruded. Their extrusion direction can differ from the world z-axis.

- *Line*
- *Point*
- *3DFace*
- *Polyline* (3D)
- *Vertex* (3D)
- *Polymesh*
- *Polyface*
- *Viewport*

These entities are planar in nature. All points are expressed in object coordinates. All of these entities can be extruded. Their extrusion direction can differ from the world z-axis.

- *Circle*
- *Arc*
- *Solid*
- *Trace*
- *Text*
- *Attrib*
- *Attdef*

- *Shape*
- *Insert*
- *Polyline* (2D)
- *Vertex* (2D)
- *LWPolyline*
- *Hatch*
- *Image*

Some of a *Dimension*'s points are expressed in WCS and some in OCS.

Elevation

Elevation group code 38:

Exists only in output from versions prior to R11. Otherwise, Z coordinates are supplied as part of each of the entity's defining points.

Arbitrary Axis Algorithm

- DXF Reference for Arbitrary Axis Algorithm provided by Autodesk.

The arbitrary axis algorithm is used by AutoCAD internally to implement the arbitrary but consistent generation of object coordinate systems for all entities that use object coordinates.

Given a unit-length vector to be used as the z-axis of a coordinate system, the arbitrary axis algorithm generates a corresponding x-axis for the coordinate system. The y-axis follows by application of the **right-hand** rule.

We are looking for the arbitrary x- and y-axis to go with the normal Az (the arbitrary z-axis). They will be called Ax and Ay (using *Vec3*):

```
Az = Vec3(entity.dxf.extrusion).normalize()    # normal (extrusion) vector
if (abs(Az.x) < 1/64.) and (abs(Az.y) < 1/64.):
    Ax = Vec3(0, 1, 0).cross(Az).normalize()    # the cross-product operator
else:
    Ax = Vec3(0, 0, 1).cross(Az).normalize()    # the cross-product operator
Ay = Az.cross(Ax).normalize()
```

WCS to OCS

```
def wcs_to_ocs(point):
    px, py, pz = Vec3(point)    # point in WCS
    x = px * Ax.x + py * Ax.y + pz * Ax.z
    y = px * Ay.x + py * Ay.y + pz * Ay.z
    z = px * Az.x + py * Az.y + pz * Az.z
    return Vec3(x, y, z)
```

OCS to WCS

```

Wx = wcs_to_ocs((1, 0, 0))
Wy = wcs_to_ocs((0, 1, 0))
Wz = wcs_to_ocs((0, 0, 1))

def ocs_to_wcs(point):
    px, py, pz = Vec3(point) # point in OCS
    x = px * Wx.x + py * Wy.y + pz * Wz.z
    y = px * Wy.x + py * Wy.y + pz * Wy.z
    z = px * Wz.x + py * Wz.y + pz * Wz.z
    return Vec3(x, y, z)

```

6.4.8 DXF Units

The DXF reference has no explicit information how to handle units in DXF, any information in this section is based on experiments with BricsCAD and may differ in other CAD application, BricsCAD tries to be as compatible with AutoCAD as possible. Therefore, this information should also apply to AutoCAD.

Please open an issue on [github](#) if you have any corrections or additional information about this topic.

Length Units

Any length or coordinate value in DXF is unitless in the first place, there is no unit information attached to the value. The unit information comes from the context where a DXF entity is used. The document/modelspace get the unit information from the header variable \$INSUNITS, paperspace and block layouts get their unit information from the attribute *units*. The modelspace object has also a units property, but this value do not represent the modelspace units, this value is always set to 0 “unitless”.

Get and set document/modelspace units as enum by the *Drawing* property *units*:

```

import ezdxf
from ezdxf import units

doc = ezdxf.new()
# Set centimeter as document/modelspace units
doc.units = units.CM
# which is a shortcut (including validation) for
doc.header['$INSUNITS'] = units.CM

```

Block Units

As said each block definition can have independent units, but there is no implicit unit conversion applied, not in CAD applications and not in ezdxf.

When inserting a block reference (INSERT) into the modelspace or another block layout with different units, the scaling factor between these units **must** be applied explicit as scaling DXF attributes (xscale,...) of the *Insert* entity, e.g. modelspace in meters and block in centimeters, x-, y- and z-scaling has to be 0.01:

```

doc.units = units.M
my_block = doc.blocks.new('MYBLOCK')
my_block.units = units.CM
block_ref = msp.add_block_ref('MYBLOCK')
# Set uniform scaling for x-, y- and z-axis
block_ref.set_scale(0.01)

```

Use helper function `conversion_factor()` to calculate the scaling factor between units:

```
factor = units.conversion_factor(doc.units, my_block.units)
# factor = 100 for 1m is 100cm
# scaling factor = 1 / factor
block_ref.set_scale(1.0/factor)
```

Hint: It is never a good idea to use different measurement system in one document, ask the NASA about their Mars Climate Orbiter from 1999. The same applies for units of the same measurement system, just use one unit like meters or inches.

Angle Units

Angles are always in degrees (360 deg = full circle) and in counter clockwise orientation, unless stated explicit otherwise.

Display Format

How values are shown in the CAD GUI is controlled by the header variables \$LUNITS and \$AUNITS, but this has no meaning for values stored in DXF files.

\$INSUNITS

The most important setting is the header variable \$INSUNITS, this variable defines the drawing units for the modelspace and therefore for the DXF document if no further settings are applied.

The modelspace LAYOUT entity has a property `units` as any layout like object, but it seem to have no meaning for the modelspace, BricsCAD set this property always to 0, which means unitless.

The most common units are 6 for meters and 1 for inches.

Changed in version 0.17.2: added an enumeration `ezdxf.enums.InsertUnits`

```
doc.header['$INSUNITS'] = 6
```

0	Unitless
1	Inches, units.IN
2	Feet, units.FT
3	Miles, units.MI
4	Millimeters, units.MM
5	Centimeters, units.CM
6	Meters, units.M
7	Kilometers, units.KM
8	Microinches
9	Mils
10	Yards, units.YD
11	Angstroms
12	Nanometers
13	Microns
14	Decimeters, units.DM
15	Decameters
16	Hectometers
17	Gigameters
18	Astronomical units
19	Light years
20	Parsecs
21	US Survey Feet
22	US Survey Inch
23	US Survey Yard
24	US Survey Mile

\$MEASUREMENT

The header variable \$MEASUREMENT controls whether the current drawing uses imperial or metric hatch pattern and linetype files, this setting is not applied correct in *ezdxf* yet, but will be fixed in the future:

This setting is independent from \$INSUNITS, it is possible to set the drawing units to inch and use metric linetypes and hatch pattern.

In BricsCAD the base scaling of the linetypes is only depending from the \$MEASUREMENT value, is not relevant if \$INSUNITS is meter, centimeter, millimeter, ... and so on and the same is valid for hatch pattern.

Changed in version 0.17.2: added an enumeration `ezdxf.enums.Measurement`

```
doc.header['$MEASUREMENT'] = 1
```

0	English
1	Metric

\$LUNITS

The header variable \$LUNITS defines how CAD applications show linear values in the GUI and has no meaning for *ezdxf*:

Changed in version 0.17.2: added an enumeration `ezdxf.enums.LengthUnits`

```
doc.header['$LUNITS'] = 2
```

1	Scientific
2	Decimal (default)
3	Engineering
4	Architectural
5	Fractional

\$AUNITS

The header variable \$AUNITS defines how CAD applications show angular values in the GUI and has no meaning for *ezdxf*, DXF angles are always degrees in counter-clockwise orientation, unless stated explicit otherwise:

Changed in version 0.17.2: added an enumeration `ezdxf.enums.AngularUnits`

```
doc.header['$AUNITS'] = 0
```

0	Decimal degrees
1	Degrees/minutes/seconds
2	Grad
3	Radians

Helper Tools

```
ezdxf.units.conversion_factor(source_units:      eздxf.enums.InsertUnits,      target_units:  
                                 eздxf.enums.InsertUnits) → float
```

Returns the conversion factor to represent *source_units* in *target_units*.

E.g. millimeter in centimeter `conversion_factor(MM, CM)` returns 0.1, because 1 mm = 0.1 cm

```
ezdxf.units.unit_name(enum: int) → str
```

Returns the name of the unit enum.

```
ezdxf.units.angle_unit_name(enum: int) → str
```

Returns the name of the angle unit enum.

6.4.9 Layout Extents and Limits

The *extents* and *limits* of an layout represents borders which can be referenced by the ZOOM command or read from some header variables from the [HeaderSection](#), if the creator application maintains these values – *ezdxf* does it not automatically.

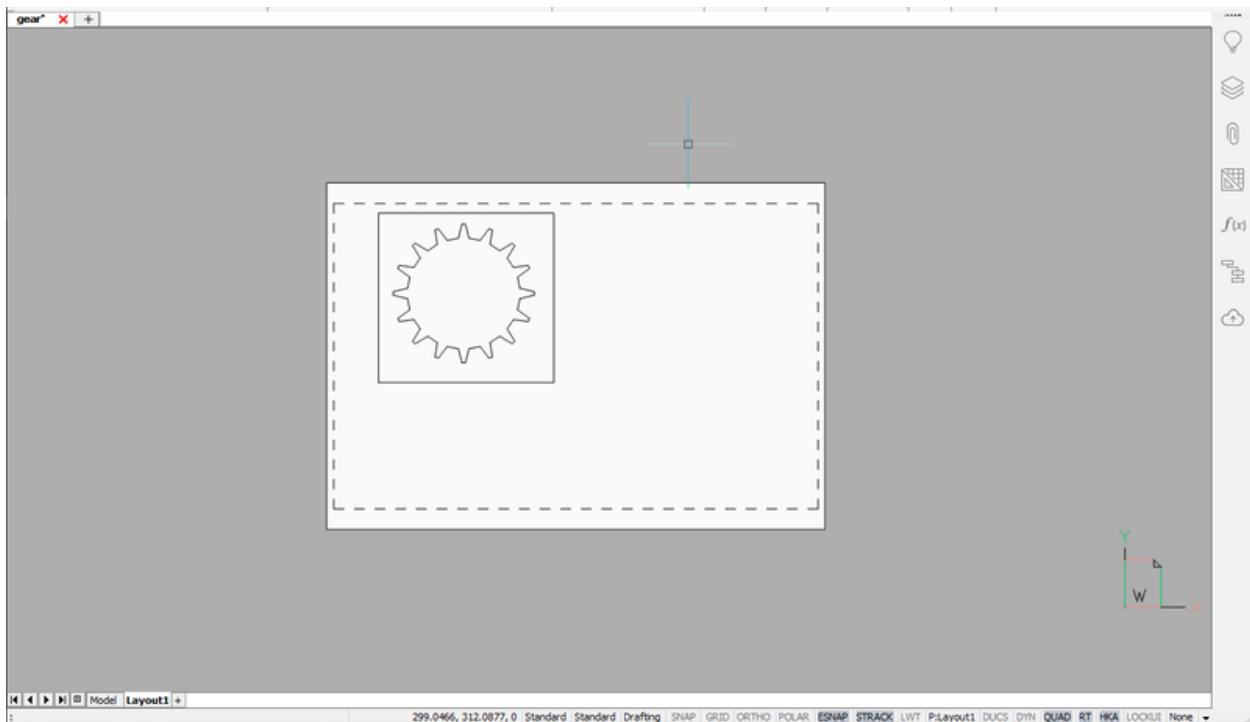
Extents

The *extents* of an layout are determined by the maximum extents of all DXF entities that are in this layout. The command:

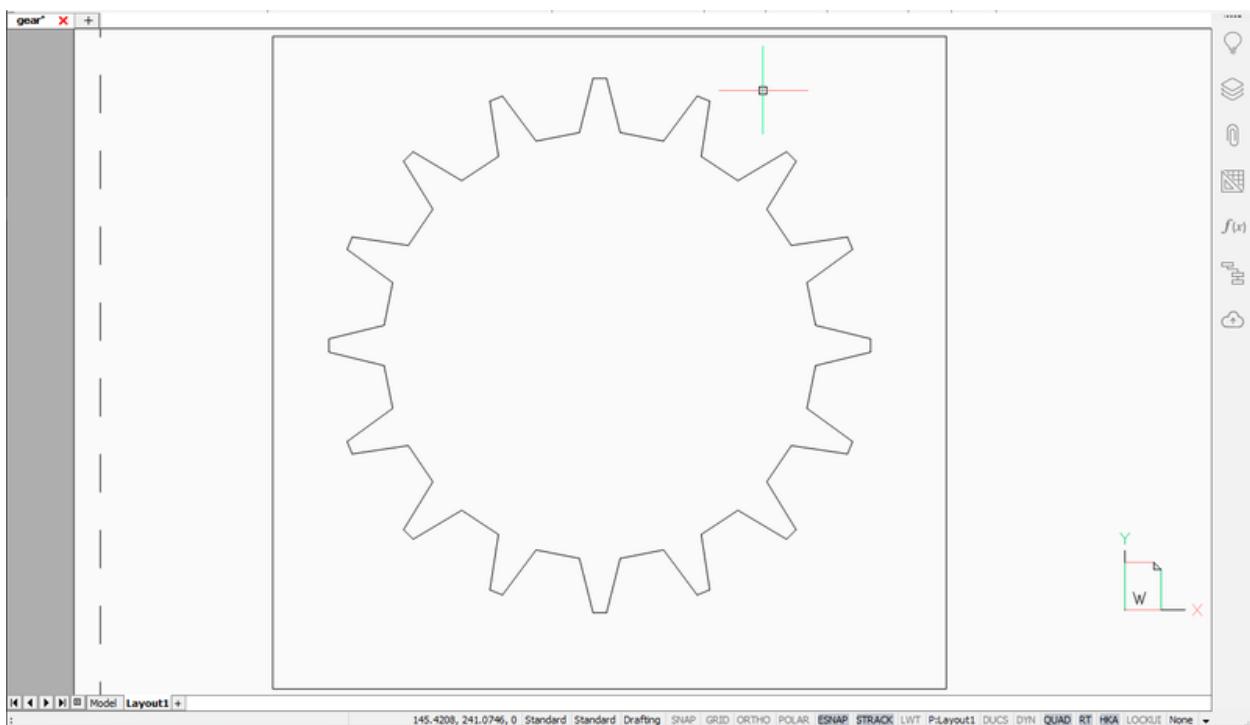
```
ZOOM extents
```

sets the current viewport to the extents of the currently selected layout.

A paper space layout in an arbitrary zoom state:



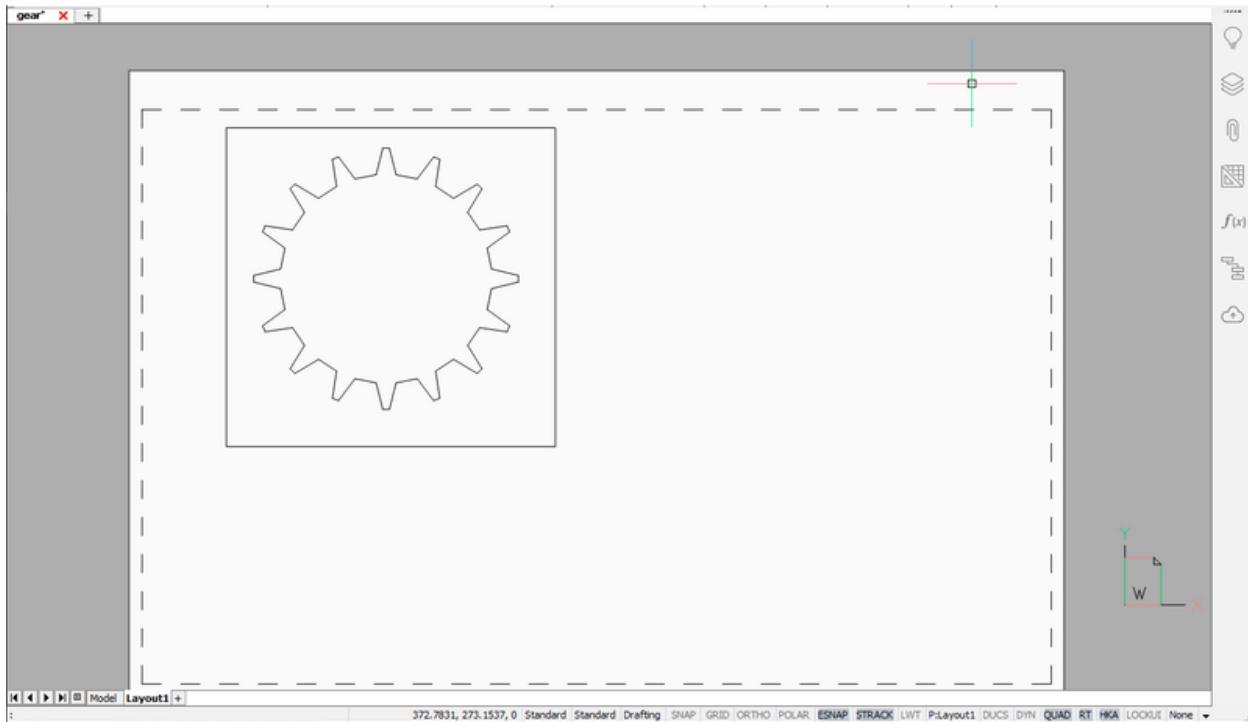
The same layout after the ZOOM extents command:



Limits

Sets an invisible rectangular boundary in the drawing area that can limit the grid display and limit clicking or entering point locations. The default limits for paper space layouts is defined by the paper size.

The layout from above after the `ZOOM all` command:



See also:

The AutoCAD online reference for the `ZOOM` and the `LIMITS` command.

Read Stored Values

The extents of the model space (the tab called “Model”) are stored in the header variable `$EXTMIN` and `$EXTMAX`. The default values of `$EXTMIN` is $(+1e20, +1e20, +1e20)$ and `$EXTMAX` is $(-1e20, -1e20, -1e20)$, which do not describe real borders. These values are copies of the `extents` attributes of the `Layout` object as `Layout.dxf.extmin` and `Layout.dxf.extmax`.

The limits of the modelspace are stored in the header variables `$LIMMIN` and `$LIMMAX` and have default values of $(0, 0)$ and $(420, 297)$, the default paper size of `ezdxf` in drawing units. These are copies of the `Layout` attributes `Layout.dxf.extmin` and `Layout.dxf.extmax`.

The extents and the limits of the *actual* paper space layout, which is the last activated paper space layout tab, stored in the header variable `$PEXTMIN`, `$PEXTMAX`, `$PLIMMIN` and `$PLIMMAX`.

Each paper space layout has its own values stored in the `Layout` attributes `Layout.dxf.extmin`, `Layout.dxf.extmax`, `Layout.dxf.limmin` and `Layout.dxf.limmax`.

Setting Extents and Limits

Since v0.16 `ezdxf` it is sufficient to define the attributes for `extents` and `limits` (`Layout.dxf.extmax`, `Layout.dxf.limmin` and `Layout.dxf.limmax`) of `Layout` object. The header variables are synchronized when the

document is saved.

The extents of a layout are not calculated automatically by *ezdxf*, as this can take a long time for large documents and correct values are not required to create a valid DXF document.

See also:

How to: *Calculate Extents for the Modelspace*

6.4.10 Font Resources

DXF relies on the infrastructure installed by AutoCAD like the included SHX files or True Type fonts. There is no simple way to store additional information about a used fonts beside the plain file system name like "arial.ttf". The CAD application or viewer which opens the DXF file has to have access to the specified fonts used in your DXF document or it has to use an appropriate replacement font, which is not that easy in the age of unicode. Later DXF versions can store font family names in the XDATA of the STYLE entity but not all CAD application use this information.

6.5 Tutorials

6.5.1 Tutorial for getting data from DXF files

In this tutorial I show you how to get data from an existing DXF drawing. If you are a new *ezdxf* user, read also the tutorial *Usage for Beginners*.

Loading the DXF file:

```
import sys
import ezdxf

try:
    doc = ezdxf.readfile("your_dxf_file.dxf")
except IOError:
    print(f"Not a DXF file or a generic I/O error.")
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f"Invalid or corrupted DXF file.")
    sys.exit(2)
```

This works well for DXF files from trusted sources like AutoCAD or BricsCAD, for loading DXF files with minor or major flaws look at the *ezdxf.recover* module.

See also:

- *Document Management*
- *Usage for Beginners*

Layouts

I use the term layout as synonym for an arbitrary entity space which can contain DXF entities like LINE, CIRCLE, TEXT and so on. Every DXF entity can only reside in exact one layout.

There are three different layout types:

- *Modelspace*: this is the common construction space

- *Paperspace*: used to create print layouts
- *BlockLayout*: reusable elements, every block has its own entity space

A DXF drawing consist of exact one modelspace and at least of one paperspace. DXF R12 has only one unnamed paperspace the later DXF versions support more than one paperspace and each paperspace has a name.

Getting the modelspace layout

The modelspace contains the “real” world representation of the drawing subjects in real world units. The modelspace has the fixed name “Model” and the DXF document has a special getter method `modelspace()`.

```
msp = doc.modelspace()
```

Iterate over DXF entities of a layout

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

```
# helper function
def print_entity(e):
    print("LINE on layer: %s\n" % e.dxf.layer)
    print("start point: %s\n" % e.dxf.start)
    print("end point: %s\n" % e.dxf.end)

# iterate over all entities in modelspace
msp = doc.modelspace()
for e in msp:
    if e.dxftype() == "LINE":
        print_entity(e)

# entity query for all LINE entities in modelspace
for e in msp.query("LINE"):
    print_entity(e)
```

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

Access DXF attributes of an entity

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

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

See *Common graphical DXF attributes*

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

```
# If DXF attribute 'paperspace' does not exist, the entity defaults
# to modelspace:
p = e.get_dxf_attrib("paperspace", 0)
```

An unsupported DXF attribute raises an `DXFAttributeError`.

Getting a paperspace layout

```
paperspace = doc.layout("layout0")
```

Retrieves the paperspace named `layout0`, the usage of the `Layout` object is the same as of the modelspace object. DXF R12 provides only one paperspace, therefore the paperspace name in the method call `doc.layout("layout0")` is ignored or can be left off. For newer DXF versions you can get a list of the available layout names by the methods `layout_names()` and `layout_names_in_taborder()`.

Retrieve entities by query language

`ezdxf` provides a flexible query language for DXF entities. All layout types have a `query()` method to start an entity query or use the `ezdxf.query.new()` function.

The query string is the combination of two queries, first the required entity query and second the optional attribute query, enclosed in square brackets: "EntityQuery [AttributeQuery]"

The entity query is a whitespace separated list of DXF entity names or the special name `*`. Where `*` means all DXF entities, all other DXF names have to be uppercase. The `*` search can exclude entity types by adding the entity name with a preceding `!` (e.g. `* !LINE`, search all entities except lines).

The attribute query is used to select DXF entities by its DXF attributes. The attribute query is an addition to the entity query and matches only if the entity already match the entity query. The attribute query is a boolean expression, supported operators: `and`, `or`, `!`.

See also:

[Entity Query String](#)

Get all LINE entities from the modelspace:

```
msp = doc.modelspace()
lines = msp.query("LINE")
```

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

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

The `*` is a wildcard for all DXF types, in this case you could also use `LINE` instead of `*`, `*` works here because `lines` just contains entities of DXF type `LINE`.

All together as one query:

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

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

```
lines_and_circles = doc.entities.query('LINE CIRCLE[layer=="construction"]')
```

Get all modelspace entities at layer construction, but excluding entities with linetype DASHED:

```
not_dashed_entities = msp.query('*[layer=="construction" and linetype!="DASHED"]')
```

Retrieve entities by groupby() function

Search and group entities by a user defined criteria. As example let's group all entities from modelspace by layer, the result will be a dict with layer names as dict-key and a list of all entities from modelspace matching this layer as dict-value. Usage as dedicated function call:

```
from eздxf.groupby import groupby
group = groupby(entities=msp, dxffattrib="layer")
```

The *entities* argument can be any container or generator which yields *DXFEntity* or inherited objects. Shorter and simpler to use as method of *BaseLayout* (modelspace, paperspace layouts, blocks) and query results as *EntityQuery* objects:

```
group = msp.groupby(dxffattrib="layer")

for layer, entities in group.items():
    print(f'Layer "{layer}" contains following entities:')
    for entity in entities:
        print(f"    {entity}")
    print("-" * 40)
```

The previous example shows how to group entities by a single DXF attribute, but it is also possible to group entities by a custom key, to do so create a custom key function, which accepts a DXF entity as argument and returns a hashable value as dict-key or None to exclude the entity. The following example shows how to group entities by layer and color, so each result entry has a tuple (*layer*, *color*) as key and a list of entities with matching DXF attributes as values:

```
def layer_and_color_key(entity):
    # return None to exclude entities from result container
    if entity.dxf.layer == "0": # exclude entities from default layer "0"
        return None
    else:
        return entity.dxf.layer, entity.dxf.color

group = msp.groupby(key=layer_and_color_key)
for key, entities in group.items():
    print(f'Grouping criteria "{key}" matches following entities:')
    for entity in entities:
        print(f"    {entity}")
    print("-" * 40)
```

To exclude entities from the result container the *key* function should return None. The *groupby()* function catches DXFAttributeError exceptions while processing entities and excludes this entities from the result container. So there is no need to worry about DXF entities which do not support certain attributes, they will be excluded automatically.

See also:

[groupby\(\)](#) documentation

6.5.2 Tutorial for creating simple DXF drawings

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

See also:

r12writer

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

```
import ezdxf

# Create a new DXF R2010 drawing, official DXF version name: "AC1024"
doc = ezdxf.new('R2010')

# Add new entities to the modelspace:
msp = doc.modelspace()
# Add a LINE entity
msp.add_line((0, 0), (10, 0))
doc.saveas('line.dxf')
```

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

See also:

Thematic Index of Layout Factory Methods

6.5.3 Tutorial for Layers

If you are not familiar with the concept of layers, please read this first: [Layer Concept](#)

Create a Layer Definition

```
import ezdxf

doc = ezdxf.new(setup=True) # setup required line types
msp = doc.modelspace()
doc.layers.add(name="MyLines", color=7, linetype="DASHED", {})
```

The advantage of assigning a linetype and a color to a layer is that entities on this layer can inherit this properties by using "BYLAYER" as linetype string and 256 as color, both values are default values for new entities so you can leave off these assignments:

```
msp.add_line((0, 0), (10, 0), dxftattribs={"layer": "MyLines"})
```

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

Changing Layer State

Get the layer definition object:

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

Check the state of the layer:

```
my_lines.is_off() # True if layer is off
my_lines.is_on() # True if layer is on
my_lines.is_locked() # True if layer is locked
layer_name = my_lines.dxf.name # get the layer name
```

Change the state of the layer:

```
# switch layer off, entities at this layer will not shown in CAD applications/viewers
my_lines.off()

# lock layer, entities at this layer are not editable in CAD applications
my_lines.lock()
```

Get/set default color of a layer by property `Layer.color`, because the DXF attribute `Layer.dxf.color` is misused for switching the layer on and off, layer is off if the color value is negative.

Changing the default layer values:

```
my_lines.dxf.linetype = "DOTTED"
my_lines.color = 13 # preserves on/off state of layer
```

See also:

For all methods and attributes see class `Layer`.

Check Available Layers

The layers object supports some standard Python protocols:

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

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

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

Deleting a Layer

Delete a layer definition:

```
doc.layers.remove("MyLines")
```

This just deletes the layer definition, all DXF entities with the DXF attribute `layer` set to "MyLines" are still there, but if they inherit color and/or linetype from the layer definition they will be drawn now with linetype "Continuous" and color 1.

6.5.4 Tutorial for Blocks

What are Blocks?

Blocks are collections of DXF entities which can be placed multiple times as block references in different layouts and other block definitions. The block reference (`Insert`) can be rotated, scaled, placed in 3D space by `OCS` and arranged in a grid like manner, each `Insert` entity can have individual attributes (`Attrib`) attached.

Create a Block

Blocks are managed as `BlockLayout` objects by the `BlocksSection` object, every drawing has only one blocks section stored in the attribute: `Drawing.blocks`.

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

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

# Create a new drawing in the DXF format of AutoCAD 2010
doc = ezdxf.new('R2010')

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

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

Block References (Insert)

A block reference is a DXF `Insert` entity and can be placed in any layout: `Modelspace`, any `Paperspace` or `BlockLayout` (which enables nested block references). Every block reference can be placed, scaled and rotated individually. Scaling by negative values is mirroring.

Lets insert some random flags into the modelspace:

```
# Get the modelspace of the drawing.
msp = doc.modelspace()

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

for point in placing_points:
    # Every flag has a different scaling and a rotation of -15 deg.
    random_scale = 0.5 + random.random() * 2.0
    # Add a block reference to the block named 'FLAG' at the coordinates 'point'.
    msp.add_blockref('FLAG', point, dxftattribs={
        'xscale': random_scale,
        'yscale': random_scale,
        'rotation': -15
    })

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

Query all block references of block FLAG:

```
for flag_ref in msp.query('INSERT[name=="FLAG"]'):
    print(str(flag_ref))
```

When inserting a block reference into the modelspace or another block layout with different units, the scaling factor between these units should be applied as scaling attributes (`xscale`, ...) e.g. modelspace in meters and block in centimeters, `xscale` has to be 0.01.

What are Attributes?

An attribute (*Attrib*) is a text annotation attached to a block reference with an associated tag. Attributes are often used to add information to blocks which can be evaluated and exported by CAD programs. An attribute can be visible or hidden. The simple way to use attributes is just to add an attribute to a block reference by `Insert.add_attrib()`, but the attribute is geometrically not related to the block reference, so you have to calculate the insertion point, rotation and scaling of the attribute by yourself.

Using Attribute Definitions

The second way to use attributes in block references is a two step process, first step is to create an attribute definition (template) in the block definition, the second step is adding the block reference by `Layout.add_blockref()` and attach and fill attribute automatically by the `add_auto_attribs()` method to the block reference. The advantage of this method is that all attributes are placed relative to the block base point with the same rotation and scaling as the block, but has the disadvantage that non uniform scaling is not handled very well. The method `Layout.add_auto_blockref()` handles non uniform scaling better by wrapping the block reference and its attributes into an anonymous block and let the CAD application do the transformation work which will create correct graphical representations at least by AutoCAD and BricsCAD. This method has the disadvantage of a more complex evaluation of attached attributes

Using attribute definitions (`Attdef`):

```
# Define some attributes for the block 'FLAG', placed relative
# to the base point, (0, 0) in this case.
flag.add_attdef('NAME', (0.5, -0.5), dxftattribs={'height': 0.5, 'color': 3})
flag.add_attdef('XPOS', (0.5, -1.0), dxftattribs={'height': 0.25, 'color': 4})
flag.add_attdef('YPOS', (0.5, -1.5), dxftattribs={'height': 0.25, 'color': 4})

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

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

    # Every flag has a different scaling and a rotation of +15 deg.
    random_scale = 0.5 + random.random() * 2.0
    blockref = msp.add_blockref('FLAG', point, dxftattribs={
        'rotation': 15
    }).set_scale(random_scale)
    blockref.add_auto_attribs(values)
```

(continues on next page)

(continued from previous page)

```
# Save the drawing.
doc.saveas("auto_blockref_tutorial.dxf")
```

Get/Set Attributes of Existing Block References

See the howto: [Get/Set Block Reference Attributes](#)

Evaluate Wrapped Block References

As mentioned above evaluation of block references wrapped into anonymous blocks is complex:

```
# Collect all anonymous block references starting with '*U'
anonymous_block_refs = modelspace.query('INSERT[name ? "^\*U.+"]')

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

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

print(flag_numbers)
```

Exploding Block References

This is an advanced feature and because *ezdxf* is still not a CAD application, the results may no be perfect. **Non uniform scaling** lead to incorrect results for text entities (TEXT, MTEXT, ATTRIB) and some other entities like HATCH with arc or ellipse path segments.

By default the “exploded” entities are added to the same layout as the block reference is located.

```
for flag_ref in msp.query('INSERT[name=="FLAG"]'):
    flag_ref.explode()
```

Examine Entities of Block References

If you just want to examine the entities of a block reference use the `virtual_entities()` method. This methods yields “virtual” entities with attributes identical to “exploded” entities but they are not stored in the entity database, have no handle and are not assigned to any layout.

```
for flag_ref in msp.query('INSERT[name=="FLAG"]'):
    for entity in flag_ref.virtual_entities():
        if entity.dxftype() == "LWPOLYLINE":
            print(f"Found {str(entity)}")
```

6.5.5 Tutorial for LWPolyline

The *LWPolyline* is defined as a single graphic entity, which differs from the old-style *Polyline* entity, which is defined as a group of sub-entities. *LWPolyline* display faster (in AutoCAD) and consume less disk space, it is a planar element, therefore all points in *OCS* as (x, y) tuples (*LWPolyline.dxf.elevation* is the z-axis value).

Create a simple polyline:

```
import ezdxf

doc = ezdxf.new("R2000")
msp = doc.modelspace()

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

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

Append multiple points to a polyline:

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

line = msp.query("LWPOLYLINE").first
if line is not None:
    line.append_points([(8, 7), (10, 7)])

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

The *LWPOLYLINE* entity always returns polyline points as 5-tuple (x, y, start_width, end_width, bulge), the start_width, end_width and bulge values are 0 if not present:

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

Use the method *points()* as context manager to edit polyline points, this method was introduced because accessing single points was very slow in early versions of *ezdxf*, but now direct access by the index operator [] is very fast and using the context manager is not required anymore. The advantage of the context manager is the ability to use a user defined point format:

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

line = msp.query("LWPOLYLINE").first

with line.points("xyseb") as points:
    # points is a standard Python list
    # existing points are 5-tuples, but new points can be
    # set as (x, y, [start_width, [end_width, [bulge]]]) tuple
```

(continues on next page)

(continued from previous page)

```
# set start_width, end_width to 0 to be ignored (x, y, 0, 0, bulge).

# delete last 2 points
del points[-2:]
# adding two points
points.extend([(4, 7), (0, 7)])

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

Each line segment can have a different start- and end-width, if omitted start- and end-width is 0:

```
doc = ezdxf.new("R2000")
msp = doc.modelspace()

# point format = (x, y, [start_width, [end_width, [bulge]]])
# set start_width, end_width to 0 to be ignored (x, y, 0, 0, bulge).

points = [(0, 0, .1, .15), (3, 0, .2, .25), (6, 3, .3, .35), (6, 6)]
msp.add_lwpolyline(points)

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

The first point carries the start- and end-width of the first segment, the second point of the second segment and so on, the start- and end-width value of the last point is used for the closing segment if the polyline is closed else these values are ignored. Start- and end-width only works if the DXF attribute `dxf.const_width` is unset, to be sure delete it:

```
# no exception will be raised if const_width is already unset:
del line.dxf.const_width
```

LWPolyline can also have curved elements, they are defined by the *Bulge value*:

```
doc = ezdxf.new("R2000")
msp = doc.modelspace()

# point format = (x, y, [start_width, [end_width, [bulge]]])
# set start_width, end_width to 0 to be ignored (x, y, 0, 0, bulge).

points = [(0, 0, 0, .05), (3, 0, .1, .2, -.5), (6, 0, .1, .05), (9, 0)]
msp.add_lwpolyline(points)

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



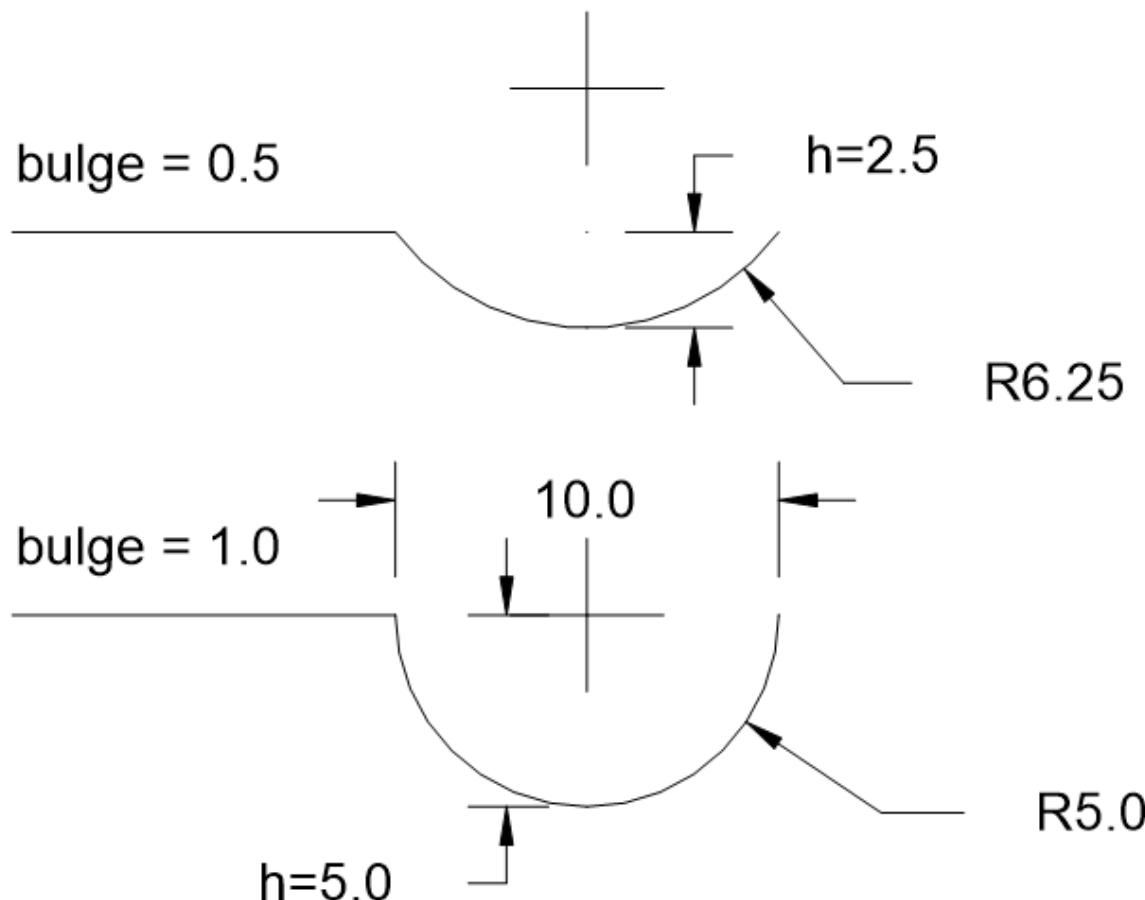
The curved segment is drawn from the point which defines the *bulge* value to the following point, the curved segment is always an arc. The bulge value defines the ratio of the arc sagitta (h) to half line segment length

(point distance), a bulge value of 1 defines a semicircle. The curve is on the right side of the line for a bulge value > 0, and on the left side of the line for a bulge value < 0.

The user defined point format, default is `xyseb`:

- `x` = x coordinate
- `y` = y coordinate
- `s` = start width
- `e` = end width
- `b` = bulge value
- `v` = (x, y) as tuple

```
msp.add_lwpolyline([(0, 0, 0), (10, 0, 1), (20, 0, 0)], format="xyb")
msp.add_lwpolyline([(0, 10, 0), (10, 10, .5), (20, 10, 0)], format="xyb")
```



6.5.6 Tutorial for Text

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

```

import ezdxf

# TEXT is a basic entity and is supported by every DXF version.
# Argument setup=True for adding standard linetypes and text styles.
doc = ezdxf.new('R12', setup=True)
msp = doc.modelspace()

# use set_pos() for proper TEXT alignment:
# The relations between DXF attributes 'halign', 'valign',
# 'insert' and 'align_point' are tricky.
msp.add_text("A Simple Text").set_pos((2, 3), align='MIDDLE_RIGHT')

# Using a text style
msp.add_text("Text Style Example: Liberation Serif",
            dxftattribs={
                'style': 'LiberationSerif',
                'height': 0.35
            }.set_pos((2, 6), align='LEFT')

doc.saveas("simple_text.dxf")

```

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

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

Special alignments are `ALIGNED` and `FIT`, they require a second alignment point, the text is justified with the vertical alignment `Baseline` on the virtual line between these two points.

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

Standard Text Styles

Setup some standard text styles and linetypes by argument `setup=True`:

```
doc = ezdxf.new('R12', setup=True)
```

Replaced all proprietary font declarations in `setup_styles()` (`ARIAL`, `ARIAL_NARROW`, `ISOCPEUR` and `TIMES`) by open source fonts, this is also the style name (e.g. `{'style': 'OpenSans-Italic'}`):

LiberationMono-Italic
LiberationMono-BoldItalic
LiberationMono-Bold
LiberationMono
LiberationSerif-Italic
LiberationSerif-BoldItalic
LiberationSerif-Bold
LiberationSerif
LiberationSans-Italic
LiberationSans-BoldItalic
LiberationSans-Bold
LiberationSans
OpenSansCondensed-Italic
OpenSansCondensed-Light
OpenSansCondensed-Bold
OpenSans-ExtraBoldItalic
OpenSans-ExtraBold
OpenSans-BoldItalic
OpenSans-Bold
OpenSans-SemiBoldItalic
OpenSans-SemiBold
OpenSans-Italic
OpenSans
OpenSans-Light-Italic
OpenSans-Light
STANDARD

New Text Style

Creating a new text style is simple:

```
doc.styles.new('myStandard', dxfattribs={'font' : 'OpenSans-Regular.ttf'})
```

But getting the correct font name is often not that simple, especially on Windows. This shows the required steps to get the font name for *Open Sans*:

- open font folder *c:\windows\fonts*
- select and open the font-family *Open Sans*
- right-click on *Open Sans Standard* and select *Properties*
- on top of the first tab you see the font name: 'OpenSans-Regular.ttf'

The style name has to be unique in the DXF document, else *ezdxf* will raise an `DXFTableEntryError` exception. To replace an existing entry, delete the existing entry by `doc.styles.remove(name)`, and add the replacement entry.

3D Text

It is possible to place the 2D `Text` entity into 3D space by using the *OCS*, for further information see: [Tutorial for OCS/UCS Usage](#).

6.5.7 Tutorial for MText and MTextEditor

The `MText` entity is a multi line entity with extended formatting possibilities and requires at least DXF version R2000, to use all features (e.g. background fill) DXF R2007 is required.

Prolog code:

```
import ezdxf

doc = ezdxf.new("R2007", setup=True)
msp = doc.modelspace()

lorem_ipsum = """
Lorem ipsum dolor sit amet, consectetur adipiscing elit,
sed do eiusmod tempor incididunt ut labore et dolore magna
aliqua. Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit
esse cillum dolore eu fugiat nulla pariatur. Excepteur sint
occaecat cupidatat non proident, sunt in culpa qui officia
deserunt mollit anim id est laborum.
"""
```

Adding a MText entity

The `MText` entity can be added to any layout (modelspace, paperspace or block) by the `add_mtext()` function.

```
# store MText entity for additional manipulations
mtext = msp.add_mtext(lorem_ipsum, dxfattribs={"style": "OpenSans"})
```

This adds a MText entity with text style “OpenSans”. The MText content can be accessed by the `text` attribute, this attribute can be edited like any Python string:

```
mttext.text += "Append additional text to the MText entity."  
# even shorter with __iadd__() support:  
mttext += "Append additional text to the MText entity."
```

 Lorem ipsum dolor sit amet, consectetur adipiscing elit,
 sed do eiusmod tempor incididunt ut labore et dolore magna
 aliqua. Ut enim ad minim veniam, quis nostrud exercitation
 ullamco laboris nisi ut aliquip ex ea commodo consequat.
 Duis aute irure dolor in reprehenderit in voluptate velit
 esse cillum dolore eu fugiat nulla pariatur. Excepteur sint
 occaecat cupidatat non proident, sunt in culpa qui officia
 deserunt mollit anim id est laborum.
Append additional text to the MText entity.

Important: Line endings “\n” will be replaced by the MTEXT line endings “\P” at DXF export, but **not** vice versa “\P” by “\n” at DXF file loading.

Text placement

The location of the MText entity is defined by the `MText.dxf.insert` and the `MText.dxf.attachment_point` attributes. The `attachment_point` defines the text alignment relative to the `insert` location, default value is 1.

Attachment point constants defined in `ezdxf.lldxf.const`:

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

The MText entity has a method for setting `insert`, `attachment_point` and `rotation` attributes by one call: `set_location()`

Character height

The character height is defined by the DXF attribute `MText.dxf.char_height` in drawing units, which has also consequences for the line spacing of the MText entity:

```
mtext.dxf.char_height = 0.5
```

The character height can be changed inline, see also [MText formatting](#) and [MText Inline Codes](#).

Text rotation (direction)

The `MText.dxf.rotation` attribute defines the text rotation as angle between the x-axis and the horizontal direction of the text in degrees. The `MText.dxf.text_direction` attribute defines the horizontal direction of MText as vector in WCS or OCS, if an [OCS](#) is defined. Both attributes can be present at the same entity, in this case the `MText.dxf.text_direction` attribute has the higher priority.

The `MText` entity has two methods to get/set rotation: `get_rotation()` returns the rotation angle in degrees independent from definition as angle or direction, and `set_rotation()` set the `rotation` attribute and removes the `text_direction` attribute if present.

Defining a wrapping border

The wrapping border limits the text width and forces a line break for text beyond this border. Without attribute `dxf.width` (or setting 0) the lines are wrapped only at the regular line endings " \P" or "\n", setting the reference column width forces additional line wrappings at the given width. The text height can not be limited, the text always occupies as much space as needed.

```
mtext.dxf.width = 60
```

```

Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt ut
labore et dolore magna
aliqua. Ut enim ad minim veniam,
quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea
commodo consequat.
Duis aute irure dolor in
reprehenderit in voluptate velit
esse cillum dolore eu fugiat nulla
pariatur. Excepteur sint
occaecat cupidatat non proident,
sunt in culpa qui officia
deserunt mollit anim id est laborum.
Append additional text to the MText
entity.
```

MText formatting

`MText` supports inline formatting by special codes: [MText Inline Codes](#)

```
mtext.text = "{\C1;red text} - {\C3;green text} - {\C5;blue text}"
```

red text - green text - blue text

See also new section for the new support class *MTextEditor* in *ezdxf* v0.17.

Stacked text

MText also supports stacked text:

```
# the space ' ' in front of 'Lower' and the ';' behind 'Lower' are necessary
# combined with vertical center alignment
mtext.text = "\\"A1;\\$Upper^ Lower; - \\$Upper/ Lower;{ - \\$Upper# Lower;"
```

The diagram illustrates stacked text. The word "Upper" is displayed in a large, bold, black font. It appears in three different configurations: once above "Lower" (top), once below "Lower" (middle), and once to the left of "Lower" (bottom). In the middle configuration, the "Upper" text is underlined. In the bottom configuration, a diagonal line passes through the "Upper" text, indicating it is part of a stack.

See also new section for the new support class *MTextEditor* in *ezdxf* v0.17.

Background color (filling)

The MText entity can have a background filling:

- AutoCAD Color Index (ACI)
- true color value as (r, g, b) tuple
- color name as string, use special name 'canvas' to use the canvas background color

Because of the complex dependencies *ezdxf* provides a method to set all required DXF attributes at once:

```
mtext.set_bg_color(2, scale=1.5)
```

The parameter *scale* determines how much border there is around the text, the value is based on the text height, and should be in the range of 1 - 5, where 1 fits exact the MText entity.

MTextEditor

New in version 0.17.

Warning: The MTextEditor assembles just the inline code, which has to be parsed and rendered by the target CAD application, *ezdxf* has no influence to that result.

Keep inline formatting as simple as possible, don't test the limits of its capabilities, this will not work across different CAD applications and keep the formatting in a logic manner like, do not change paragraph properties in the middle of a paragraph.

There is no official documentation for the inline codes!

The `MTextEditor` class provides a floating interface to build `MText` content in an easy way.

This example only shows the connection between `MText` and the `MTextEditor`, and shows no additional features to the first example of this tutorial:

Init Editor

```
import eздxf
from eздxf.tools.text import MTextEditor

doc = eздxf.new("R2007", setup=True)
msp = doc.modelspace()

lorem_ipsum = """
Lorem ipsum dolor sit amet, consectetur adipiscing elit, ... see prolog code
"""

# create a new editor object with an initial text:
editor = MTextEditor(lorem_ipsum)

# get the MTEXT content string from the editor by the str() function:
mtext = msp.add_mtext(str(editor), dxftattribs={"style": "OpenSans"})
```

Tutorial Prolog:

```
# use constants defined in MTextEditor:
NP = MTextEditor.NEW_PARAGRAPH

ATTRIBS = {
    "char_height": 0.7,
    "style": "OpenSans",
    "width": 10,
}
editor = MTextEditor("using colors:" + NP)
```

Set Text Color

There are three ways to change the color inline:

- by color name “red”, “green”, “blue”, “yellow”, “cyan”, “magenta”, “white”
- by *AutoCAD Color Index (ACI)*
- by RGB values

```
# RED: set color by name - red, green, blue, yellow, cyan, magenta, white
editor.color("red").append("RED" + NP)
# RED: the color stays the same until the next change
editor.append("also RED" + NP)

# GREEN: change color by ACI (AutoCAD Color Index)
editor.aci(3).append("GREEN" + NP)

# BLUE: change color by RGB tuples
editor.rgb((0, 0, 255)).append("BLUE" + NP)

# add the MTEXT entity to the model space:
msp.add_mtext(str(editor), attribs)
```



Changing Text Height

The `MtextEditor.height()` method set the text height as absolute value in drawing units (text height = cap height):

```
attribs = dict(ATTRIBS)
attribs["width"] = 40.0
editor = MTextEditor("changing text height absolute: default height is 0.7" + NP)
# doubling the default height = 1.4
editor.height(1.4)
editor.append("text height: 1.4" + NP)
editor.height(3.5).append("text height: 3.5" + NP)
editor.height(0.7).append("back to default height: 0.7" + NP)
msp.add_mtext(str(editor), attribs)
```



The `MtextEditor.scale_height()` method set the text height by a relative factor, the `MtextEditor` object does not keep track of current text height, you have to do this by yourself. The initial text height is `MText.dxf.char_height`:

```
attribs = dict(ATTRIBS)
attribs["width"] = 40.0
editor = MTextEditor("changing text height relative: default height is 0.7" + NP)
# this is the default text height in the beginning:
current_height = attribs["char_height"]
# The text height can only be changed by a factor:
editor.scale_height(2) # scale by 2 = 1.4
# keep track of the actual height:
current_height *= 2
editor.append("text height: 1.4" + NP)
# to set an absolute height, calculate the required factor:
desired_height = 3.5
factor = desired_height / current_height
editor.scale_height(factor).append("text height: 3.5" + NP)
current_height = desired_height
# and back to 0.7
editor.scale_height(0.7 / current_height).append("back to default height: 0.7" + NP)
msp.add_mtext(str(editor), attribs).set_location(insert=location)
```

Changing Font

The font name for changing MText fonts inline is the font family name! The font family name is the name shown in font selection widgets in desktop applications: “Arial”, “Times New Roman”, “Comic Sans MS”. The font has to be installed at the target system, else then CAD default font will be used, in AutoCAD/BricsCAD is this the font defined for the text style “Standard”.

Important: The DXF/DWG format is not optimal for preserving text layouts across multiple systems, and it's getting really bad across different CAD applications.

```
attribs = dict(ATTRIBS)
attribs["width"] = 15.0
editor = MTextEditor("changing fonts:" + NP)
editor.append("Default: Hello World!" + NP)
editor.append("SimSun: ")
# change font in a group to revert back to the default font at the end:
simsun_editor = MTextEditor().font("SimSun").append("!" + NP)
# reverts the font back at the end of the group:
editor.group(str(simsun_editor))
# back to default font OpenSans:
editor.append("Times New Roman: ")
# change font outside of a group until next font change:
editor.font("Times New Roman").append(" !" + NP)
# If the font does not exist, a replacement font will be used:
editor.font("Does not exist").append("This is the replacement font!")
msp.add_mtext(str(editor), attribs)
```

changing fonts:
 Default: Hello World!
 SimSun: 你好，世界
 Times New Roman: Привет мир!
 This is the replacement
 font!

Set Paragraph Properties

The paragraph properties are set by the `paragraph()` method and a `ParagraphProperties` object, which bundles all paragraph properties in a named tuple.

Each paragraph can have its own properties for:

- indentation arguments:
 - `indent` is the left indentation of the first line
 - `left` is the left side indentation of the paragraph
 - `right` is the right side indentation of the paragraph
- text adjustment: `align`, by enum `MTextParagraphAlignment`
 - `MTextParagraphAlignment.LEFT`
 - `MTextParagraphAlignment.RIGHT`
 - `MTextParagraphAlignment.CENTER`
 - `MTextParagraphAlignment.JUSTIFIED`
 - `MTextParagraphAlignment.DISTRIBUTED`
- tabulator stops: `tab_stops`, a tuple of tabulator stops

Indentation and tabulator stops are multiples of the default `MText` text height stored in `MText.dxf.char_height`. Calculate the drawing units for indentation and tabulator stops, by multiplying the the indentation value by the `char_height` value.

`Mtext` paragraphs are separated by new paragraph “P” characters.

```
# import support classes:
from ezdxf.tools.text import ParagraphProperties, MTextParagraphAlignment

ATTRIBS = {
    'align': MTextParagraphAlignment.CENTER,
    'left': 1000,
    'right': 1000,
    'tab_stops': [(1000, 1000), (2000, 1000), (3000, 1000)]
}
```

(continues on next page)

(continued from previous page)

```

attribs["char_height"] = 0.25
attribs["width"] = 7.5
editor = MTextEditor("Indent the first line:" + NP)
props = ParagraphProperties(
    indent=1, # indent first line = 1x0.25 drawing units
    align=MTextParagraphAlignment.JUSTIFIED
)
editor.paragraph(props)
editor.append(lorem_ipsum)
msp.add_mtext(str(editor), attribs)

```

Indent the first line:

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

The first line indentation “indent” is relative to the “left” indentation.

```

# import support classes:
from ezdxf.tools.text import ParagraphProperties, MTextParagraphAlignment

attribs = dict(ATTRIBS)
attribs["char_height"] = 0.25
attribs["width"] = 7.5
editor = MTextEditor("Indent left paragraph side:" + NP)
indent = 0.7 # 0.7 * 0.25 = 0.175 drawing units
props = ParagraphProperties(
    # first line indentation is relative to "left", this reverses the
    # left indentation:
    indent=-indent, # first line
    # indent left paragraph side:
    left=indent,
    align=MTextParagraphAlignment.JUSTIFIED
)
editor.paragraph(props)
editor.append(" ".join(lorem_ipsum(100)))
msp.add_mtext(str(editor), attribs).set_location(insert=location)

```

Indent left paragraph side:

 Lorem ipsum dolor sit amet, consetetur
 sadipscing elitr, sed diam nonumy eirmod
 tempor invidunt ut labore et dolore
 magna aliquyam erat, sed diam voluptua.
 At vero eos et accusam et justo duo
 dolores et ea rebum. Stet clita kasd
 gubergren, no sea takimata sanctus est
 Lorem ipsum dolor sit amet. Lorem ipsum
 dolor sit amet, consetetur sadipscing elitr,
 sed diam nonumy eirmod tempor
 invidunt ut labore et dolore magna
 aliquyam erat, sed diam voluptua. At vero
 eos et accusam et justo duo dolores et ea
 rebum. Stet clita kasd gubergren, no sea
 takimata sanctus est Lorem ipsum dolor
 sit amet.

Bullet List

There are no special commands to build bullet list, the list is build of indentation and a tabulator stop. Each list item needs a marker as an arbitrary string. For more information about paragraph indentation and tabulator stops see also chapter *Set Paragraph Properties*.

```
attribs = dict(ATTRIBS)
attribs["char_height"] = 0.25
attribs["width"] = 7.5
bullet = ". " # alt + numpad 7
editor = MTextEditor("Bullet List:" + NP)
editor.bullet_list(
    indent=1,
    bullets=[bullet] * 3, # each list item needs a marker
    content=[
        "First item",
        "Second item",
        " ".join(lorem_ipsum(30)),
    ])
msp.add_mtext(str(editor), attribs)
```

Bullet List:

- First item
- Second item
- Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et

Numbered List

There are no special commands to build numbered list, the list is build of indentation and a tabulator stop. There is no automatic numbering, but therefore the absolute freedom for using any string as list marker. For more information about paragraph indentation and tabulator stops see also chapter [Set Paragraph Properties](#).

```
attribs = dict(ATTRIBS)
attribs["char_height"] = 0.25
attribs["width"] = 7.5
editor = MTextEditor("Numbered List:" + NP)
editor.bullet_list(
    indent=1,
    bullets=["1.", "2.", "3."],
    content=[
        "First item",
        "Second item",
        " ".join(lorem_ipsum(30)),
    ])
msp.add_mtext(str(editor), attribs)
```

Numbered List:

1. First item
2. Second item
3. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et

Stacked Text

MText supports stacked text (fractions) as a single inline code, which means it is not possible to change any property inside the fraction. This example shows a fraction with scaled down text height, placed in a group to revert the text height afterwards:

```
editor = MTextEditor("Stacked text:" + NP)

stack = MTextEditor().scale_height(0.6).stack("1", "2", "^")
editor.append("over: ").group(str(stack)).append(NP)

stack = MTextEditor().scale_height(0.6).stack("1", "2", "/")
editor.append("fraction: ").group(str(stack)).append(NP)

stack = MTextEditor().scale_height(0.6).stack("1", "2", "#")
editor.append("slanted: ").group(str(stack)).append(NP)

# Additional formatting in numerator and denominator is not supported
# by AutoCAD or BricsCAD, switching the color inside the stacked text
# to red does not work:
numerator = MTextEditor().color("red").append("1")
stack = MTextEditor().scale_height(0.6).stack(str(numerator), "2", "#")
editor.append("color red: ").group(str(stack)).append(NP)

msp.add_mtext(str(editor), attribs)
```



See also:

- `MTextEditor` example code on [github](#).
- Documentation of `MTextEditor`

6.5.8 Tutorial for Spline

Background information about B-spline at [Wikipedia](#).

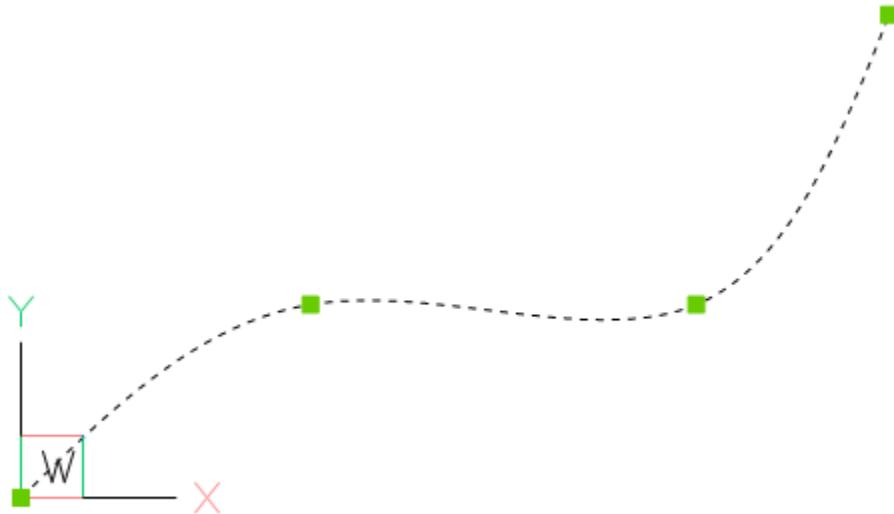
Splines from fit points

Splines can be defined by fit points only, this means the curve goes through all given fit points. AutoCAD and BricsCAD generates required control points and knot values by itself, if only fit points are present.

Create a simple spline:

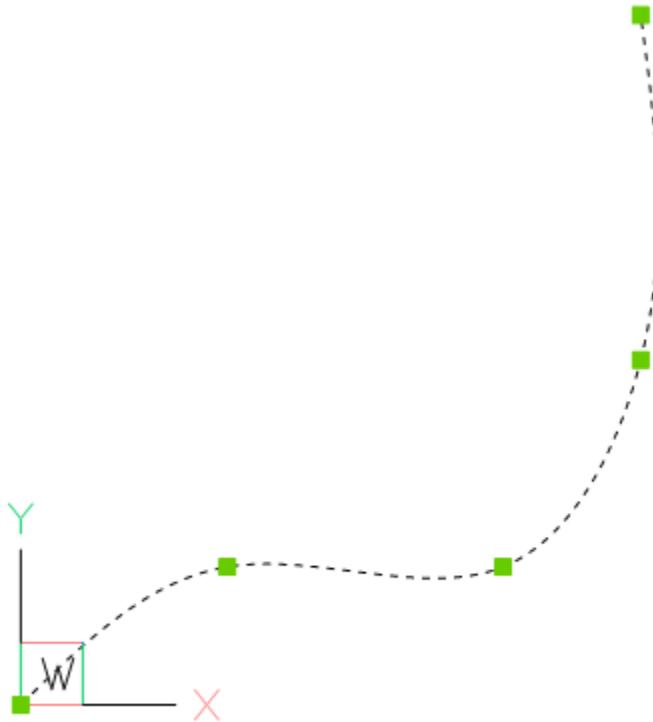
```
doc = ezdxf.new('R2000')

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



Append a fit point to a spline:

```
# fit_points, control_points, knots and weights are list-like containers:
spline.fit_points.append((2250, 2500, 0))
```



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

Solve problems of incorrect values after editing a spline generated by AutoCAD:

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

msp = doc.modelspace()
spline = msp.query('SPLINE').first

# fit_points, control_points, knots and weights are list-like objects:
spline.fit_points.append((2250, 2500, 0))
```

As far as I have tested, this approach works without complaints from AutoCAD, but for the case of problems remove invalid data:

```
# current control points do not match spline defined by fit points
spline.control_points = []

# count of knots is not correct:
# count of knots = count of control points + degree + 1
spline.knots = []

# same for weights, count of weights == count of control points
spline.weights = []
```

Splines by control points

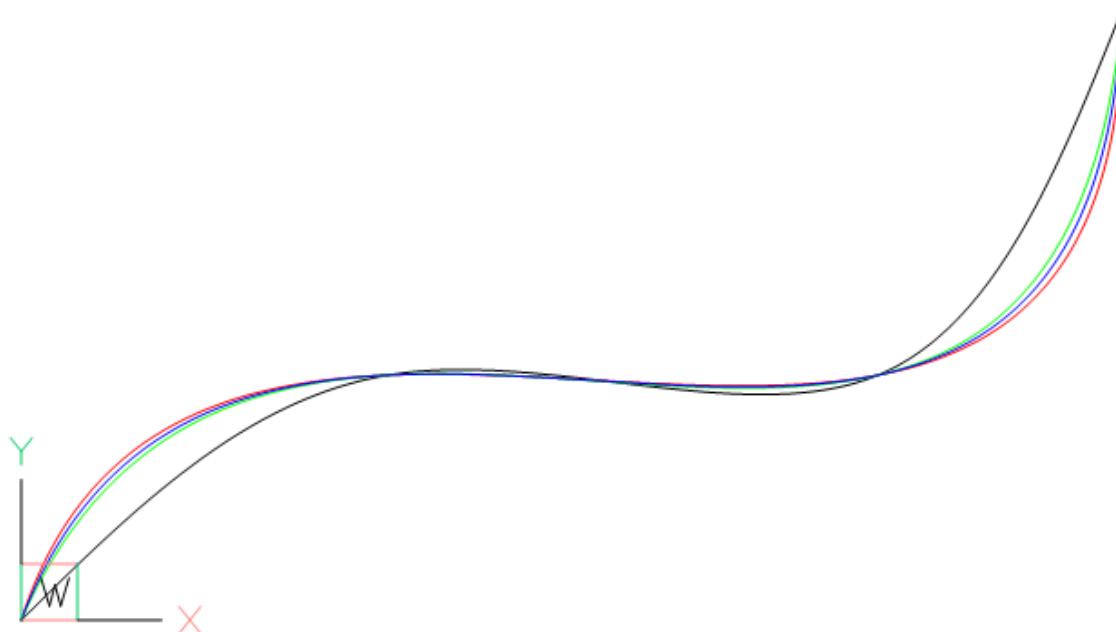
To create splines from fit points is the easiest way to create splines, but this method is also the least accurate, because a spline is defined by control points and knot values, which are generated for the case of a definition by fit points, and the worst fact is that for every given set of fit points exist an infinite number of possible splines as solution.

AutoCAD (and BricsCAD also) uses an proprietary algorithm to generate control points and knot values from fit points, which differs from the well documented [Global Curve Interpolation](#). Therefore splines generated from fit points by `ezdxf` do not match splines generated by AutoCAD (BricsCAD).

To ensure the same spline geometry for all CAD applications, the spline has to be defined by control points. The method `add_spline_control_frame()` adds a spline trough fit points by calculating the control points by the [Global Curve Interpolation](#) algorithm. There is also a low level function `ezdxf.math.global_bspline_interpolation()` which calculates the control points from fit points.

```
msp.add_spline_control_frame(fit_points, method='uniform', dxfattribs={'color': 1})
msp.add_spline_control_frame(fit_points, method='chord', dxfattribs={'color': 3})
msp.add_spline_control_frame(fit_points, method='centripetal', dxfattribs={'color': 5}
    ↵)
```

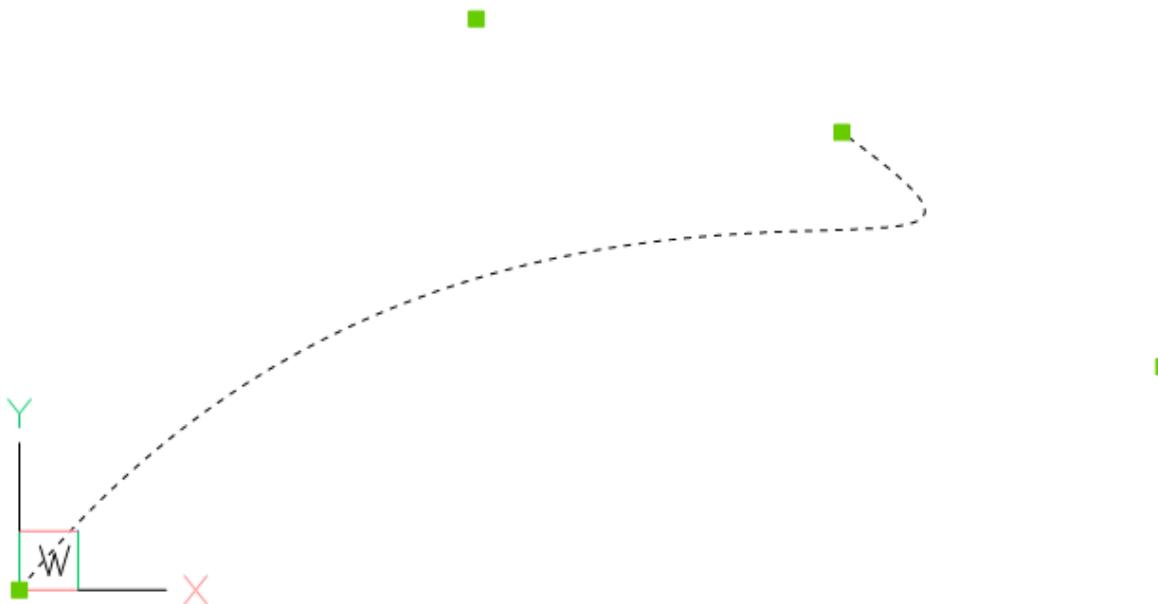
- black curve: AutoCAD/BricsCAD spline generated from fit points
- red curve: spline curve interpolation, “uniform” method
- green curve: spline curve interpolation, “chord” method
- blue curve: spline curve interpolation, “centripetal” method



Open Spline

Add and open (clamped) spline defined by control points with the method `add_open_spline()`. If no knot values are given, an open uniform knot vector will be generated. A clamped B-spline starts at the first control point and ends at the last control point.

```
control_points = [(0, 0, 0), (1250, 1560, 0), (3130, 610, 0), (2250, 1250, 0)]
msp.add_open_spline(control_points)
```

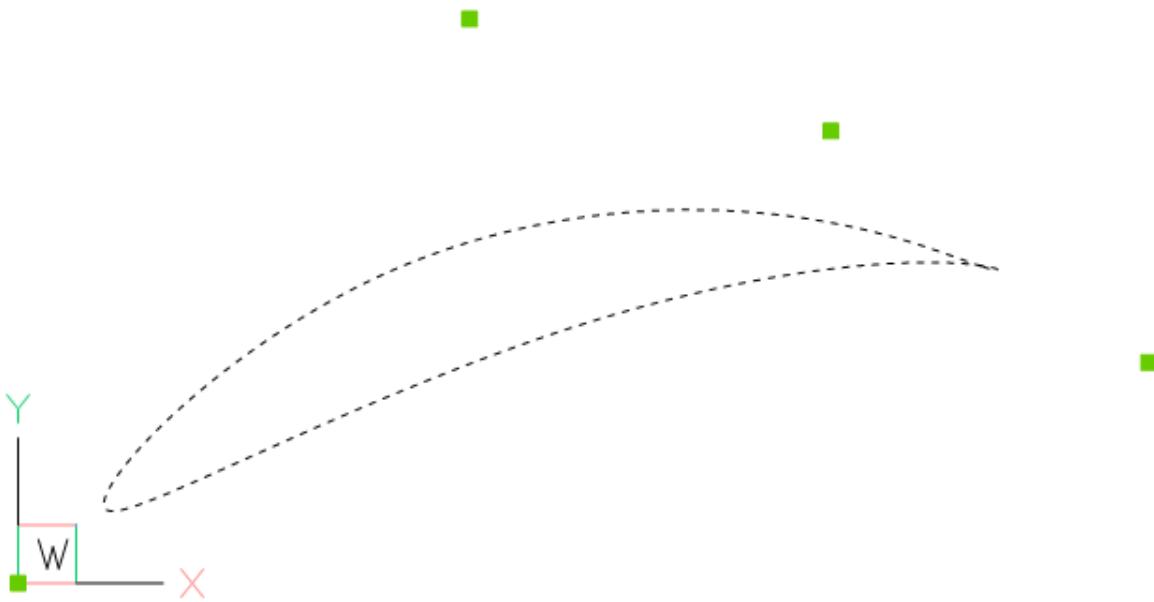


Rational Spline

Rational B-splines have a weight for every control point, which can raise or lower the influence of the control point, default weight = 1, to lower the influence set a weight < 1 to raise the influence set a weight > 1. The count of weights has to be always equal to the count of control points.

Example to raise the influence of the first control point:

```
msp.add_closed_rational_spline(control_points, weights=[3, 1, 1, 1])
```



Spline properties

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

```
if spline.closed:  
    # this spline is closed  
    pass  
  
# close spline  
spline.closed = True  
  
# open spline  
spline.closed = False
```

Set start- and end tangent for splines defined by fit points:

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

Get data count as stored in DXF file:

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

Get data count of real existing data:

```
count = spline.fit_point_count  
count = spline.control_point_count  
count = spline.knot_count
```

6.5.9 Tutorial for Polyface

coming soon ...

6.5.10 Tutorial for Mesh

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

```
import ezdxf

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

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

# MESH requires DXF R2000 or later
doc = ezdxf.new("R2000")
msp = doc.modelspace()
mesh = msp.add_mesh()
# do not subdivide cube, 0 is the default value
mesh.dxf.subdivision_levels = 0
with mesh.edit_data() as mesh_data:
    mesh_data.vertices = cube_vertices
    mesh_data.faces = cube_faces

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

Create a cube mesh by assembling single faces and the `edit_data()` context manager of the `Mesh` class, using the helper class `MeshData`:

```
import ezdxf

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

(continues on next page)

(continued from previous page)

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

# MESH requires DXF R2000 or later
doc = ezdxf.new("R2000")
msp = doc.modelspace()
mesh = msp.add_mesh()

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

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

6.5.11 Tutorial for Hatch

Create hatches with one boundary path

The simplest form of the `Hatch` entity has one polyline path with only straight lines as boundary path:

```
import ezdxf

# hatch requires DXF R2000 or later
doc = ezdxf.new("R2000")
msp = doc.modelspace()

# by default a solid fill hatch with fill color=7 (white/black)
hatch = msp.add_hatch(color=2)

# every boundary path is a 2D element
# vertex format for the polyline path is: (x, y[, bulge])
# there are no bulge values in this example
hatch.paths.add_polyline_path(
    [(0, 0), (10, 0), (10, 10), (0, 10)], is_closed=True
)

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

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

```
import ezdxf

# hatch requires the DXF R2000 or later
doc = ezdxf.new("R2000")
```

(continues on next page)

(continued from previous page)

```

msp = doc.modelspace()

# by default a solid fill hatch with fill color=7 (white/black)
hatch = msp.add_hatch(color=2)

# every boundary path is a 2D element
# vertex format for the polyline path is: (x, y[, bulge])
# bulge value 1 = an arc with diameter=10 (= distance to next vertex * bulge value)
# bulge value > 0 ... arc is right of line
# bulge value < 0 ... arc is left of line
hatch.paths.add_polyline_path(
    [(0, 0, 1), (10, 0), (10, 10, -0.5), (0, 10)], is_closed=True
)

doc.saveas("solid_hatch_polyline_path_with_bulge.dxf")

```

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

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

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

```

import ezdxf

# hatch requires the DXF R2000 or later
doc = ezdxf.new("R2000")
msp = doc.modelspace()

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

# by default a solid fill hatch with fill color=7 (white/black)
hatch = msp.add_hatch(color=2)

# every boundary path is a 2D element
edge_path = hatch.paths.add_edge_path()
# each edge path can contain line, arc, ellipse and spline elements
# important: major axis >= minor axis (ratio <= 1.)
edge_path.add_ellipse((0, 0), major_axis=(0, 10), ratio=0.5)

doc.saveas("solid_hatch_ellipse.dxf")

```

Create hatches with multiple boundary paths (islands)

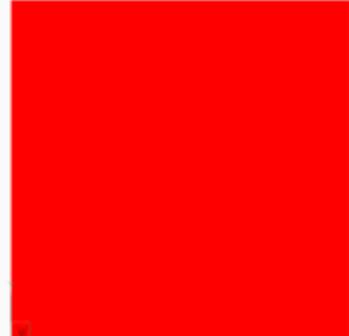
The DXF attribute `hatch_style` defines the island detection style:

0	nested - altering filled and unfilled areas
1	outer - area between <i>external</i> and <i>outermost</i> path is filled
2	ignore - <i>external</i> path is filled

```
hatch = msp.add_hatch(
    color=1,
    dxffattribs={
        "hatch_style": ezdxf.const.HATCH_STYLE_NESTED,
        # 0 = nested: ezdxf.const.HATCH_STYLE_NESTED
        # 1 = outer: ezdxf.const.HATCH_STYLE_OUTERMOST
        # 2 = ignore: ezdxf.const.HATCH_STYLE_IGNORE
    },
)

# The first path has to set flag: 1 = external
# flag const.BOUNDARY_PATH_POLYLINE is added (OR) automatically
hatch.paths.add_polyline_path(
    [(0, 0), (10, 0), (10, 10), (0, 10)],
    is_closed=True,
    flags=ezdxf.const.BOUNDARY_PATH_EXTERNAL,
)
```

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



```
# The second path has to set flag: 16 = outermost
hatch.paths.add_polyline_path(
    [(1, 1), (9, 1), (9, 9), (1, 9)],
    is_closed=True,
    flags=ezdxf.const.BOUNDARY_PATH_OUTERMOST,
)
```

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

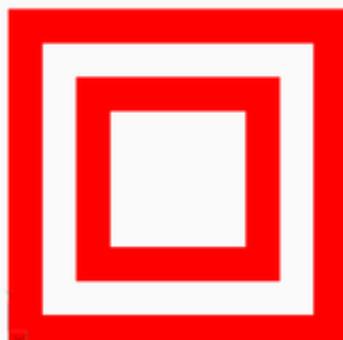


```
# The third path has to set flag: 0 = default
hatch.paths.add_polyline_path(
    [(2, 2), (8, 2), (8, 8), (2, 8)],
    is_closed=True,
    flags=ezdxf.const.BOUNDARY_PATH_DEFAULT,
)
```



```
# The forth path has to set flag: 0 = default, and so on
hatch.paths.add_polyline_path(
    [(3, 3), (7, 3), (7, 7), (3, 7)],
    is_closed=True,
    flags=ezdxf.const.BOUNDARY_PATH_DEFAULT,
)

doc.saveas(OUTDIR / "solid_hatch_islands_04.dxf")
```



The expected result of combinations of various `hatch_style` values and paths `flags`, or the handling of overlapping paths is not documented by the DXF reference, so don't ask me, ask Autodesk or just try it by yourself and post your experience in the forum.

Example for Edge Path Boundary

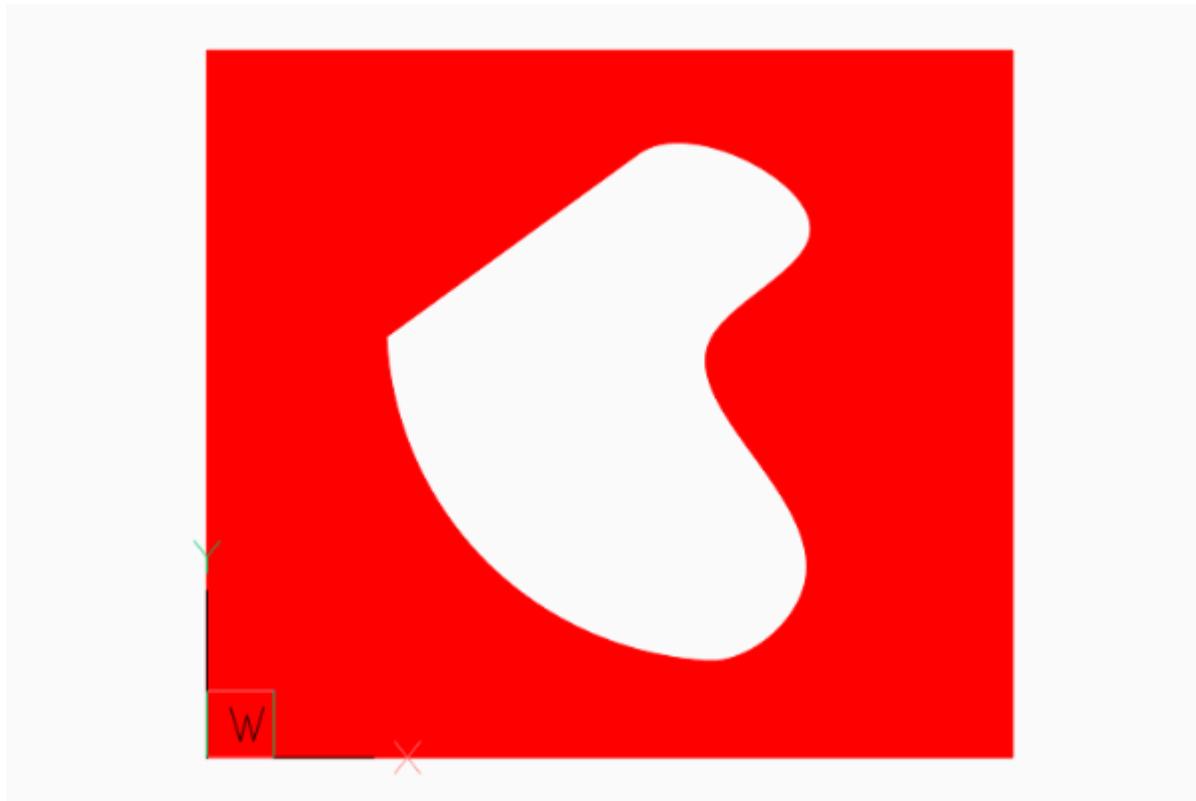
```
hatch = msp.add_hatch(color=1)

# 1. polyline path
hatch.paths.add_polyline_path(
    [
```

(continues on next page)

(continued from previous page)

```
(240, 210, 0),
(0, 210, 0),
(0, 0, 0.0),
(240, 0, 0),
],
is_closed=1,
flags=ezdxf.const.BOUNDARY_PATH_EXTERNAL,
)
# 2. edge path
edge_path = hatch.paths.add_edge_path(flags=ezdxf.const.BOUNDARY_PATH_OUTERMOST)
edge_path.add_spline(
    control_points=[
        (126.658105895725, 177.0823706957212),
        (141.5497003747484, 187.8907860433995),
        (205.8997365206943, 154.7946313459515),
        (113.0168862297068, 117.8189380884978),
        (202.9816918983783, 63.17222935389572),
        (157.363511042264, 26.4621294342132),
        (144.8204003260554, 28.4383294369643),
    ],
    knot_values=[
        0.0,
        0.0,
        0.0,
        0.0,
        55.20174685732758,
        98.33239645153571,
        175.1126541251052,
        213.2061566683142,
        213.2061566683142,
        213.2061566683142,
        213.2061566683142,
    ],
)
edge_path.add_arc(
    center=(152.6378550678883, 128.3209356351659),
    radius=100.1880612627354,
    start_angle=94.4752130054052,
    end_angle=177.1345242028005,
)
edge_path.add_line(
    (52.57506282464041, 123.3124200796114),
    (126.658105895725, 177.0823706957212),
)
```



Associative Boundary Paths

A HATCH entity can be associative to a base geometry, which means if the base geometry is edited in a CAD application the HATCH get the same modification. Because *ezdxf* is **not** a CAD application, this association is **not** maintained nor verified by *ezdxf*, so if you modify the base geometry afterwards the geometry of the boundary path is not updated and no verification is done to check if the associated geometry matches the boundary path, this opens many possibilities to create invalid DXF files: USE WITH CARE.

This example associates a LWPOLYLINE entity to the hatch created from the LWPOLYLINE vertices:

```
# Create base geometry
lwpolyline = msp.add_lwpolyline(
    [(0, 0, 0), (10, 0, 0.5), (10, 10, 0), (0, 10, 0)],
    format="xyb",
    close=True,
)

hatch = msp.add_hatch(color=1)
path = hatch.paths.add_polyline_path(
    # get path vertices from associated LWPOLYLINE entity
    lwpolyline.get_points(format="xyb"),
    # get closed state also from associated LWPOLYLINE entity
    is_closed=lwpolyline.closed,
)

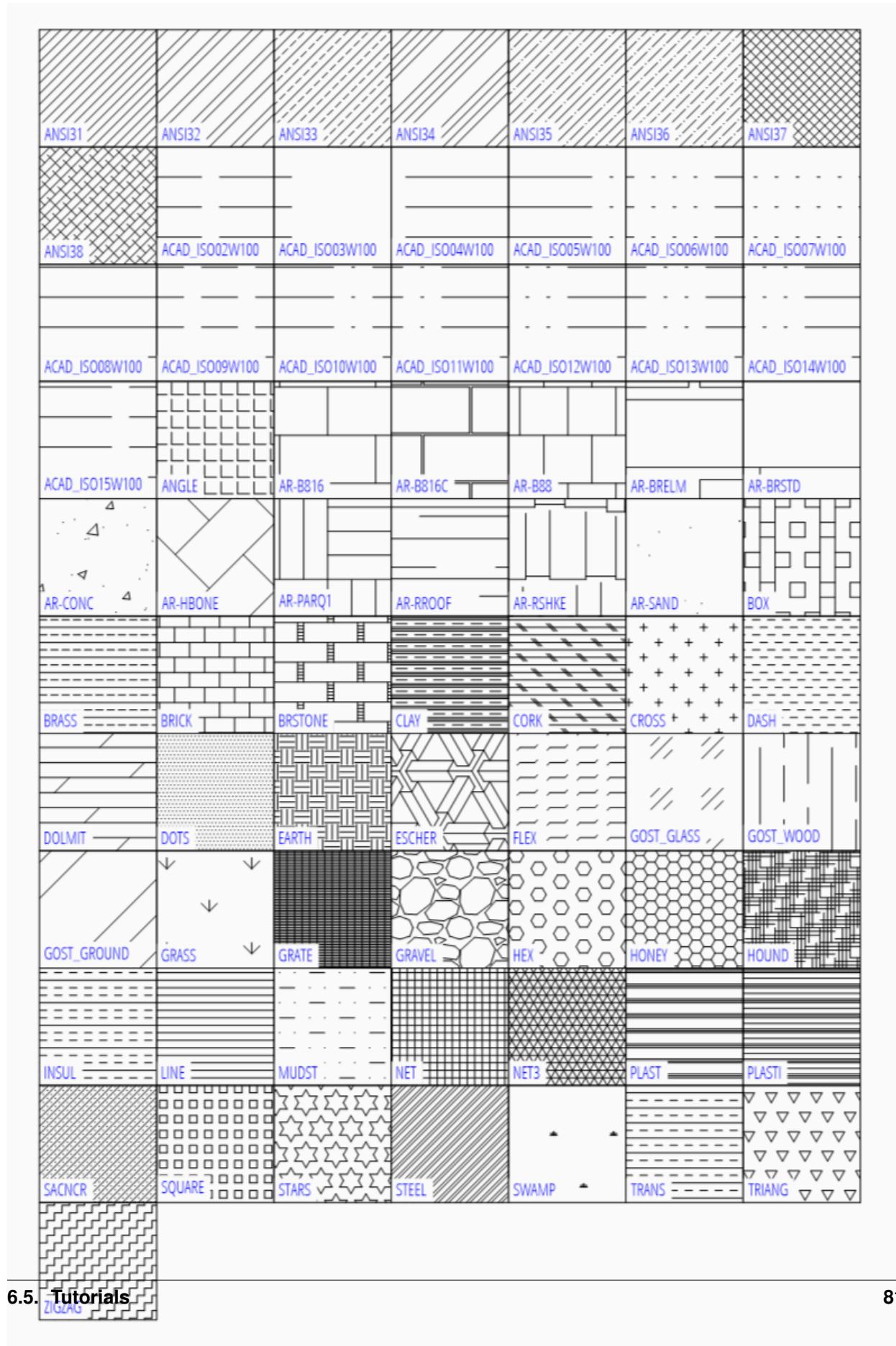
# Set association between boundary path and LWPOLYLINE
hatch.associate(path, [lwpolyline])
```

An EdgePath needs associations to all geometry entities forming the boundary path.

Predefined Hatch Pattern

Use predefined hatch pattern by name:

```
hatch.set_pattern_fill("ANSI31", scale=0.5)
```



Create hatches with gradient fill

TODO

6.5.12 Tutorial for Hatch Pattern Definition

TODO

6.5.13 Tutorial for Image and ImageDef

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

```
import ezdxf

# The IMAGE entity requires the DXF R2000 format or later.
doc = ezdxf.new("R2000")

# The IMAGEDEF entity is like a block definition, it just defines the image.
my_image_def = doc.add_image_def(
    filename="mycat.jpg", size_in_pixel=(640, 360)
)

msp = doc.modelspace()
# The IMAGE entity is like the INSERT entity, it's just an image reference,
# and there can be multiple references to the same picture in a DXF document.

# 1st image reference
msp.add_image(
    insert=(2, 1),
    size_in_units=(6.4, 3.6),
    image_def=my_image_def,
    rotation=0
)
# 2nd image reference
msp.add_image(
    insert=(4, 5),
    size_in_units=(3.2, 1.8),
    image_def=my_image_def,
    rotation=30
)

# Get existing image definitions from the OBJECTS section:
image_defs = doc.objects.query("IMAGEDEF")

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

6.5.14 Tutorial for Underlay and UnderlayDefinition

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

```
import ezdxf
```

(continues on next page)

(continued from previous page)

```

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

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

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

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

doc.saveas("dxf_with_underlay.dxf")

```

6.5.15 Tutorial for Linetypes

Simple line type example:

You can define your own linetypes. A linetype definition has a name, a description and line pattern elements:

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

total_pattern_length Sum of all linetype elements (absolute values)

elem if elem > 0 it is a line, if el

(continues on next page)

(continued from previous page)

Setup some predefined linetypes:

```
for name, desc, pattern in linetypes():
    if name not in doc.linetypes:
        doc.linetypes.add(
            name=name,
            pattern= pattern,
            description=desc,
        )
```

Check Available Linetypes

The linetypes object supports some standard Python protocols:

```
# iteration
print("available linetypes:")
for lt in doc.linetypes:
    print(f"{lt.dxf.name}: {lt.dxf.description}")

# check for existing linetype
if "DOTTED" in doc.linetypes:
    pass

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

Removing Linetypes

Warning: Ezdxf does not check if a linetype is still in use and deleting a linetype which is still in use generates an invalid DXF file. The audit process `audit()` of the DXF document removes linetype attributes referencing non existing linetypes.

You can delete a linetype:

```
doc.layers.remove("DASHED")
```

This just removes the linetype definition, the `linetype` attribute of DXF entities may still refer the removed linetype definition “DASHED” and AutoCAD will not open DXF files including undefined linetypes.

6.5.16 Tutorial for Complex Linetypes

In DXF R13 Autodesk introduced complex linetypes, containing TEXT or SHAPES in line types.

Complex linetype example with text:

Complex line type example with shapes:

For easy usage the pattern string for complex line types is mostly the same string as the pattern definition strings in AutoCAD “.lin” files.

Example for complex line type TEXT:

```
doc = ezdxf.new("R2018") # DXF R13 or later is required

doc.linetypes.add(
    name="GASLEITUNG2",
    # linetype definition string from acad.lin:
    pattern='A,.5,-.2,[ "GAS", STANDARD, S=.1, U=0.0, X=-0.1, Y=-.05],-.25',
    description= "Gasleitung2 ----GAS----GAS----GAS----GAS----GAS----",
    length=1, # required for complex line types
)}
```

The pattern always starts with an “A”, the following float values have the same meaning as for simple linetypes, a value > 0 is a line, a value < 0 is a gap, and a 0 is a point, the opening square bracket “[” starts the complex part of the linetype pattern.

The text after the “[” defines the complex linetype:

- A text in quotes (e.g. “GAS”) defines a *complex TEXT linetype* and represents the pattern text itself.
- A text without quotes is a SHAPE name (in “.lin” files) and defines a *complex SHAPE linetype*. Ezdxf can not translate this SHAPE name from the “.lin” file into the required shape file index, so *YOU have to translate this SHAPE name into the shape file index, e.g. saving the file with AutoCAD as DXF and searching for the DXF linetype definition, see example below and the DXF Internals: [LTYPE Table](#).

For *complex TEXT linetypes* the second parameter is the text style, for *complex SHAPE linetypes* the second parameter is the shape file name, the shape file has to be in the same directory as the DXF file or in one of the CAD application support paths.

The meaning of the following complex linetype parameters are shown in the table below:

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

These parameters are case insensitive and the closing square bracket “]” ends the complex part of the linetype pattern.

The fine tuning of this parameters is a try an error process, for *complex TEXT linetypes* the scaling factor (e.g. the STANDARD text style) sets the text height (e.g. “S=0.1” sets the text height to 0.1 units), by shifting in y-direction by

half of the scaling factor, the text is vertically centered to the line. For the x-direction it seems to be a good practice to place a gap in front of the text and after the text, find x shifting value and gap sizes by try and error. The overall length is at least the sum of all line and gap definitions (absolute values).

Example for complex line type SHAPE:

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

Complex line types with shapes only work if the associated shape file (e. g. ltypeshp.shx) and the DXF file are in the same directory or the shape file is placed in one of the CAD application support folders.

6.5.17 Tutorial for OCS/UCS Usage

For OCS/UCS usage is a basic understanding of vectors required, for a brush up, watch the YouTube tutorials of [3Blue1Brown](#) about [Linear Algebra](#).

Second read the [Coordinate Systems](#) introduction please.

See also:

The free online book [3D Math Primer for Graphics and Game Development](#) is a very good resource for learning vector math and other graphic related topics, it is easy to read for beginners and especially targeted to programmers.

For [WCS](#) there is not much to say as, it is what it is: the main world coordinate system, and a drawing unit can have any real world unit you want. Autodesk added some mechanism to define a scale for dimension and text entities, but because I am not an AutoCAD user, I am not familiar with it, and further more I think this is more an AutoCAD topic than a DXF topic.

Object Coordinate System (OCS)

The [OCS](#) is used to place planar 2D entities in 3D space. **ALL** points of a planar entity lay in the same plane, this is also true if the plane is located in 3D space by an OCS. There are three basic DXF attributes that gives a 2D entity its spatial form.

Extrusion

The extrusion vector defines the OCS, it is a normal vector to the base plane of a planar entity. This *base plane* is always located in the origin of the [WCS](#). But there are some entities like [Ellipse](#), which have an extrusion vector, but do not establish an OCS. For this entities the extrusion vector defines only the extrusion direction and thickness defines the extrusion distance, but all other points in WCS.

Elevation

The elevation value defines the z-axis value for all points of a planar entity, this is an OCS value, and defines the distance of the entity plane from the *base plane*.

This value exists only in output from DXF versions prior to R11 as separated DXF attribute (group code 38). In DXF R12 and later, the elevation value is supplied as z-axis value of each point. But as always in DXF, this simple rule does not apply to all entities: [LWPolyline](#) and [Hatch](#) have an DXF attribute `elevation`, where the z-axis of this point is the elevation height and the x-axis = y-axis = 0.

Thickness

Defines the extrusion distance for an entity.

Note: There is a new edition of this tutorial using UCS based transformation, which are available in *ezdxf* v0.11 and later: [Tutorial for UCS Based Transformations](#)

This edition shows the **hard way** to accomplish the transformations by low level operations.

Placing 2D Circle in 3D Space

The colors for axis follow the AutoCAD standard:

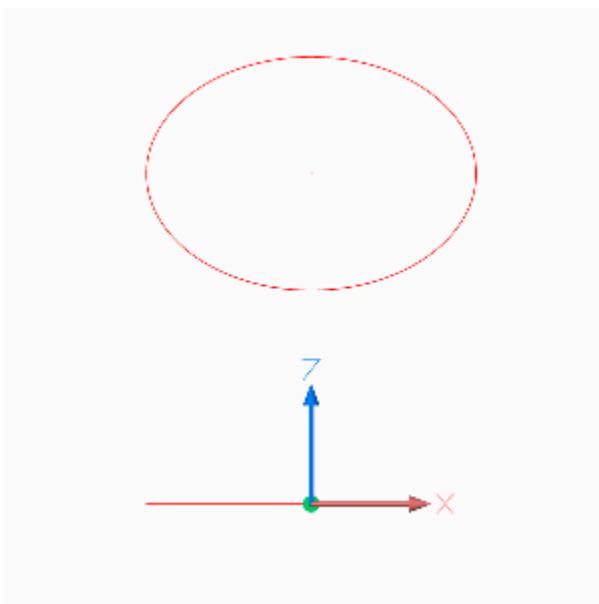
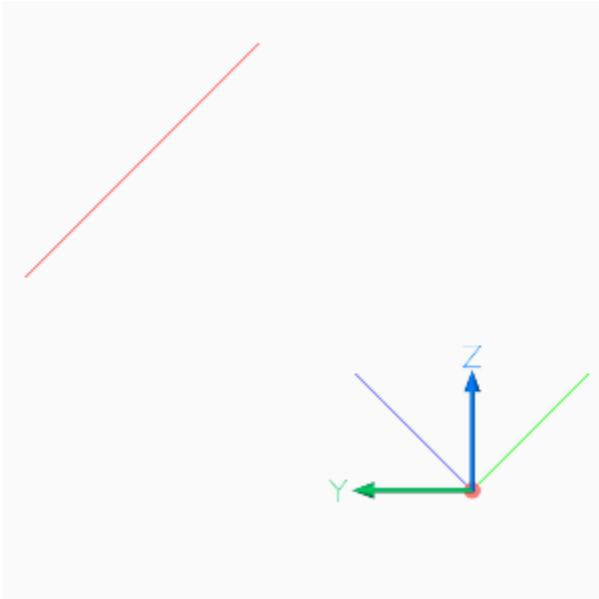
- red is x-axis
- green is y-axis
- blue is z-axis

```
import ezdxf
from ezdxf.math import OCS

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

# For this example the OCS is rotated around x-axis about 45 degree
# OCS z-axis: x=0, y=1, z=1
# extrusion vector must not normalized here
ocs = OCS((0, 1, 1))
msp.add_circle(
    # You can place the 2D circle in 3D space
    # but you have to convert WCS into OCS
    center=ocs.from_wcs((0, 2, 2)),
    # center in OCS: (0.0, 0.0, 2.82842712474619)
    radius=1,
    dxftattribs={
        # here the extrusion vector should be normalized,
        # which is granted by using the ocs.uz
        'extrusion': ocs.uz,
        'color': 1,
    })
# mark center point of circle in WCS
msp.add_point((0, 2, 2), dxftattribs={'color': 1})
```

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



Placing LWPolyline in 3D Space

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

```
# The center of the pentagon should be (0, 2, 2), and the shape is
# rotated around x-axis about 45 degree, to accomplish this I use an
# UCS with z-axis (0, 1, 1) and an x-axis parallel to WCS x-axis.
ucs = UCS(
    origin=(0, 2, 2), # center of pentagon
    ux=(1, 0, 0), # x-axis parallel to WCS x-axis
    uz=(0, 1, 1), # z-axis
)
# calculating corner points in local (UCS) coordinates
points = [Vec3.from_deg_angle((360 / 5) * n) for n in range(5)]
```

(continues on next page)

(continued from previous page)

```
# converting UCS into OCS coordinates
ocs_points = list(ucs.points_to_ocs(points))

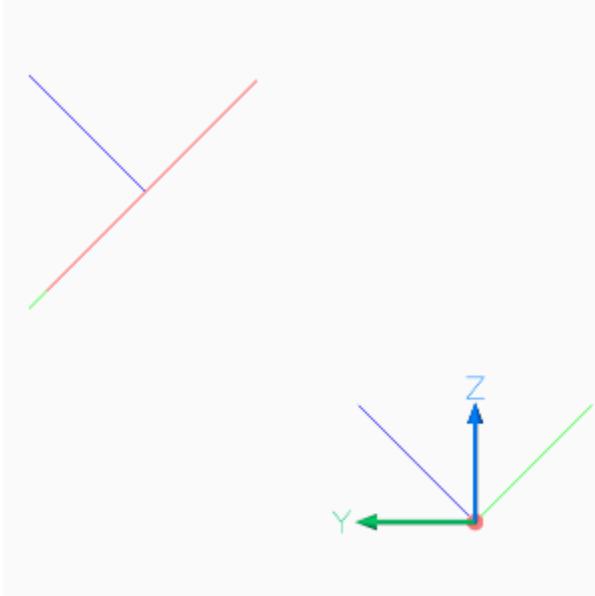
# LWPOLYLINE accepts only 2D points and has an separated DXF attribute elevation.
# All points have the same z-axis (elevation) in OCS!
elevation = ocs_points[0].z

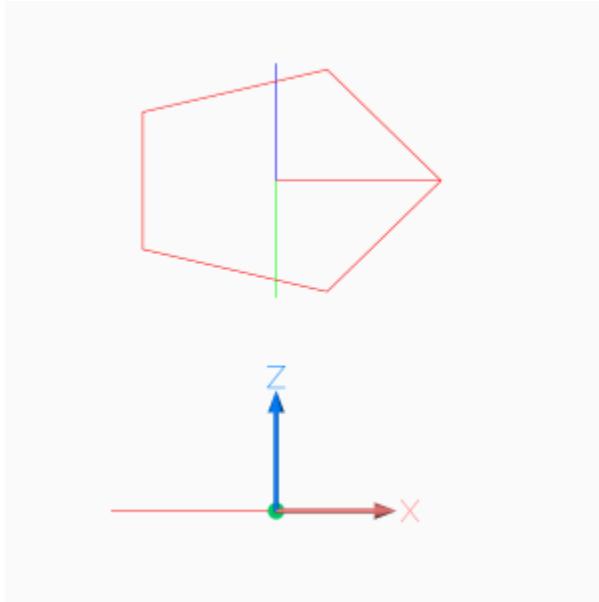
msp.add_lwpolyline(
    points=ocs_points,
    format='xy', # ignore z-axis
    close=True,
    dxftattribs={
        'elevation': elevation,
        'extrusion': ucs.uz,
        'color': 1,
    })

```

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

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





Using UCS to Place 3D Polyline

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

```
# Using an UCS simplifies 3D operations, but UCS definition can happen later
# calculating corner points in local (UCS) coordinates without Vec3 class
angle = math.radians(360 / 5)
corners_ucs = [(math.cos(angle * n), math.sin(angle * n), 0) for n in range(5)]

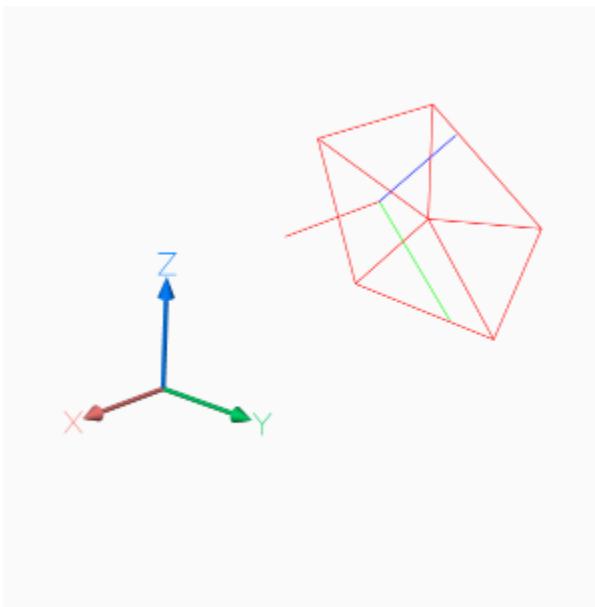
# let's do some transformations
tmatrix = Matrix44.chain( # creating a transformation matrix
    Matrix44.z_rotate(math.radians(15)), # 1. rotation around z-axis
    Matrix44.translate(0, .333, .333), # 2. translation
)
transformed_corners_ucs = tmatrix.transform_vertices(corners_ucs)

# transform UCS into WCS
ucs = UCS(
    origin=(0, 2, 2), # center of pentagon
    ux=(1, 0, 0), # x-axis parallel to WCS x-axis
    uz=(0, 1, 1), # z-axis
)
corners_wcs = list(ucs.points_to_wcs(transformed_corners_ucs))

msp.add_polyline3d(
    points=corners_wcs,
    close=True,
)

# add lines from center to corners
center_wcs = ucs.to_wcs((0, .333, .333))
for corner in corners_wcs:
    msp.add_line(center_wcs, corner, dxfattribs={'color': 1})

ucs.render_axis(msp)
```



Placing 2D Text in 3D Space

The problem by placing text in 3D space is the text rotation, which is always counter clockwise around the OCS z-axis, and 0 degree is in direction of the positive OCS x-axis, and the OCS x-axis is calculated by the [Arbitrary Axis Algorithm](#).

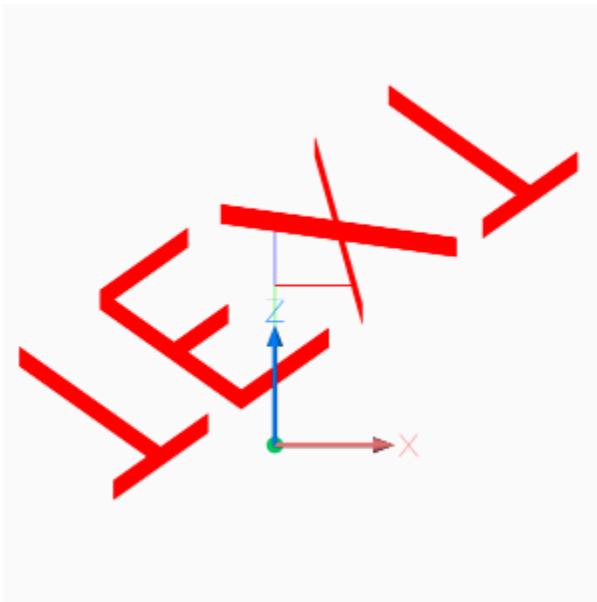
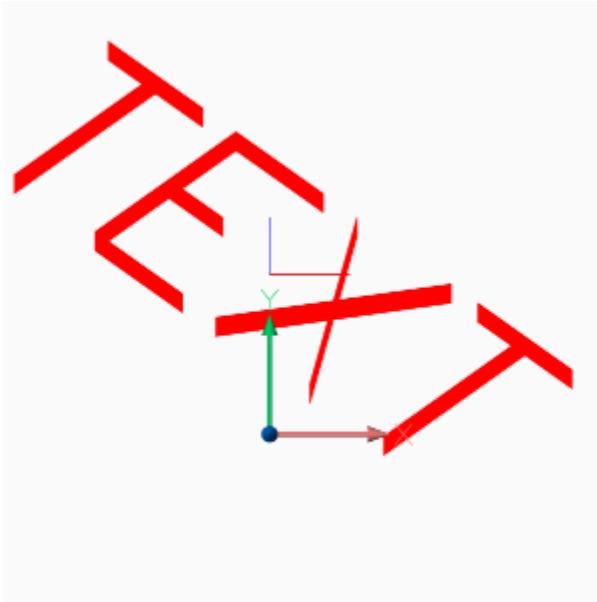
Calculate the OCS rotation angle by converting the TEXT rotation angle (in UCS or WCS) into a vector or begin with text direction as vector, transform this direction vector into OCS and convert the OCS vector back into an angle in the OCS xy-plane (see example), this procedure is available as `UCS.to_ocs_angle_deg()` or `UCS.to_ocs_angle_rad()`.

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

```
# Thickness for text works only with shx fonts not with true type fonts
doc.styles.new('TXT', dxftattribs={'font': 'romans.shx'})

ucs = UCS(origin=(0, 2, 2), ux=(1, 0, 0), uz=(0, 1, 1))
# calculation of text direction as angle in OCS:
# convert text rotation in degree into a vector in UCS
text_direction = Vec3.from_deg_angle(-45)
# transform vector into OCS and get angle of vector in xy-plane
rotation = ucs.to_ocs(text_direction).angle_deg

text = msp.add_text(
    text="TEXT",
    dxftattribs={
        # text rotation angle in degrees in OCS
        'rotation': rotation,
        'extrusion': ucs.uz,
        'thickness': .333,
        'color': 1,
        'style': 'TXT',
    })
# set text position in OCS
text.set_pos(ucs.to_ocs((0, 0, 0)), align='MIDDLE_CENTER')
```



Hint: For calculating OCS angles from an UCS, be aware that 2D entities, like TEXT or ARC, are placed parallel to the xy-plane of the UCS.

Placing 2D Arc in 3D Space

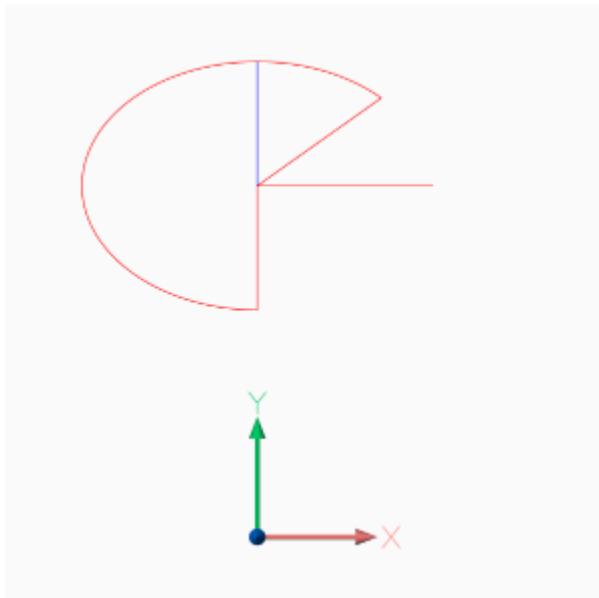
Here we have the same problem as for placing text, you need the start and end angle of the arc in degrees in OCS, and this example also shows a shortcut for calculating the OCS angles.

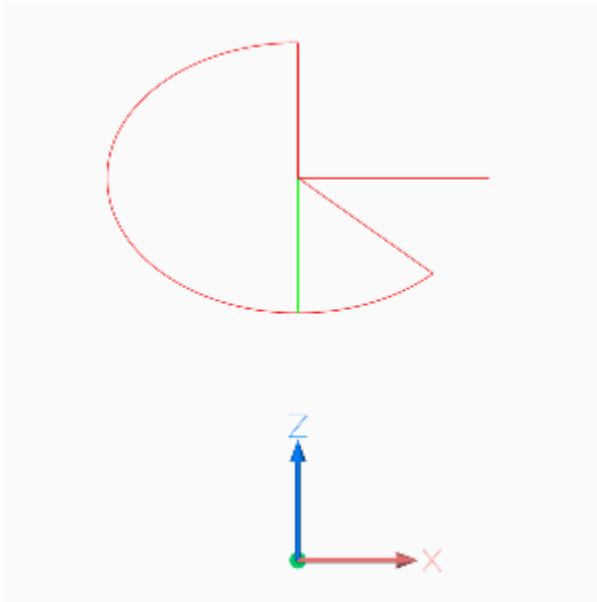
```
ucs = UCS(origin=(0, 2, 2), ux=(1, 0, 0), uz=(0, 1, 1))
msp.add_arc(
    center=ucs.to_ocs((0, 0)),
    radius=1,
```

(continues on next page)

(continued from previous page)

```
start_angle=ucs.to_ocs_angle_deg(45),
end_angle=ucs.to_ocs_angle_deg(270),
dxftattribs={
    'extrusion': ucs.uz,
    'color': 1,
})
center = ucs.to_wcs((0, 0))
msp.add_line(
    start=center,
    end=ucs.to_wcs(Vec3.from_deg_angle(45)),
    dxftattribs={'color': 1},
)
msp.add_line(
    start=center,
    end=ucs.to_wcs(Vec3.from_deg_angle(270)),
    dxftattribs={'color': 1},
)
```





Placing Block References in 3D Space

Despite the fact that block references (`Insert`) can contain true 3D entities like `Line` or `Mesh`, the `Insert` entity uses the same placing principle as `Text` or `Arc` shown in the previous chapters.

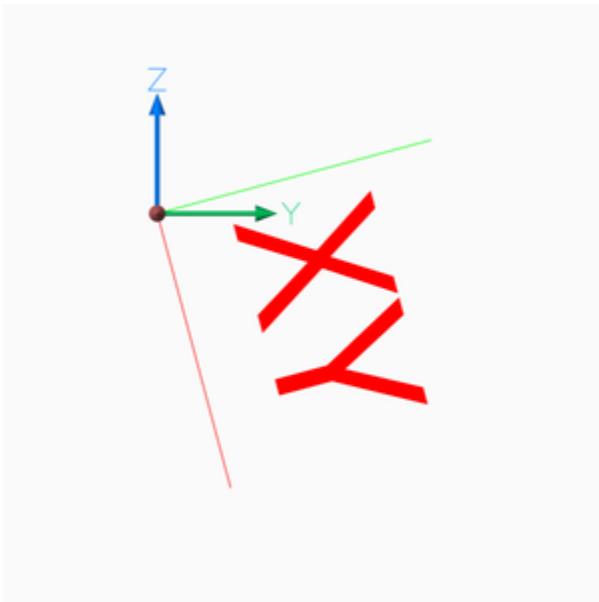
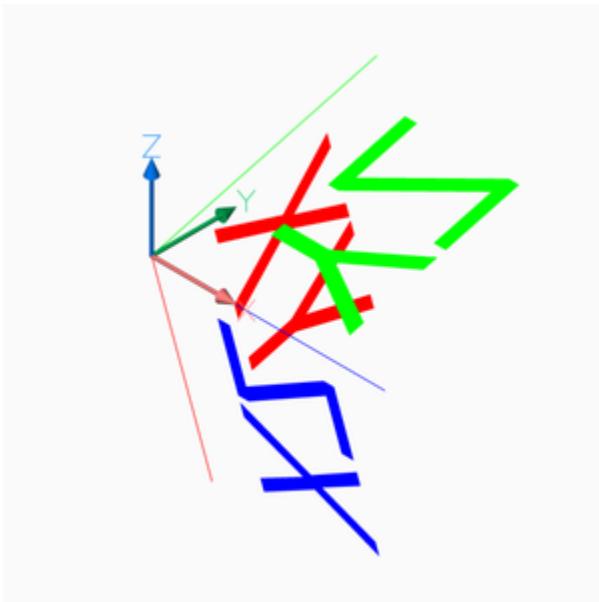
Simple placing by OCS and rotation about the z-axis, can be achieved the same way as for generic 2D entity types. The DXF attribute `Insert.dxf.rotation` rotates a block reference around the block z-axis, which is located in the `Block.dxf.base_point`. To rotate the block reference around the WCS x-axis, a transformation of the block z-axis into the WCS x-axis is required by rotating the block z-axis 90 degree counter clockwise around y-axis by using an UCS:

This is just an excerpt of the important parts, see the whole code of `insert.py` at [github](#).

```
# rotate UCS around an arbitrary axis:
def ucs_rotation(ucs: UCS, axis: Vec3, angle: float):
    # new in ezdxf v0.11: UCS.rotate(axis, angle)
    t = Matrix44.axis_rotate(axis, math.radians(angle))
    ux, uy, uz = t.transform_vertices([ucs.ux, ucs.uy, ucs.uz])
    return UCS(origin=ucs.origin, ux=ux, uy=uy, uz=uz)

doc = ezdxf.new('R2010', setup=True)
blk = doc.blocks.new('CSYS')
setup_csys(blk)
msp = doc.modelspace()

ucs = ucs_rotation(UCS(), axis=Y_AXIS, angle=90)
# transform insert location to OCS
insert = ucs.to_ocs((0, 0, 0))
# rotation angle about the z-axis (= WCS x-axis)
rotation = ucs.to_ocs_angle_deg(15)
msp.add_blockref('CSYS', insert, dxftattribs={
    'extrusion': ucs.uz,
    'rotation': rotation,
})
```



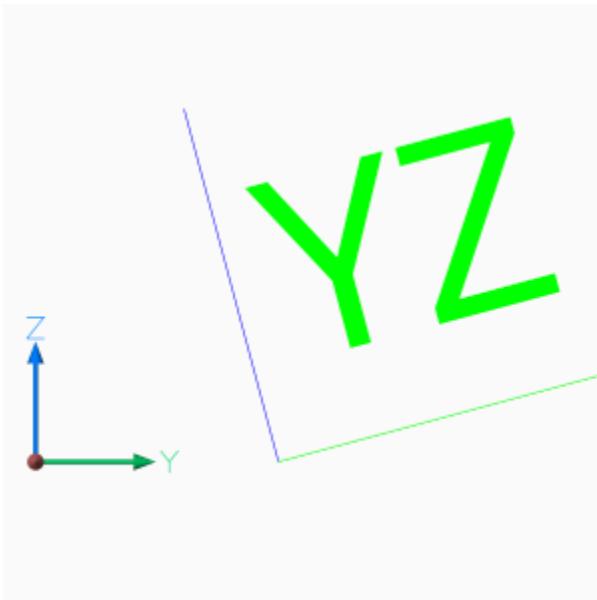
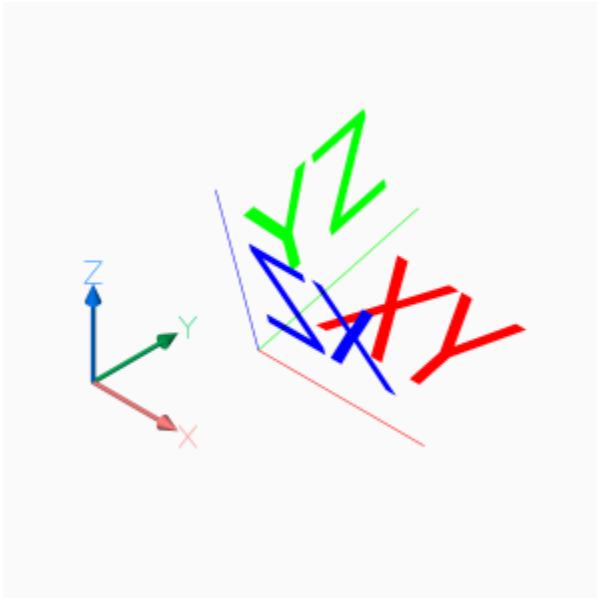
To rotate a block reference around another axis than the block z-axis, you have to find the rotated z-axis (extrusion vector) of the rotated block reference, following example rotates the block reference around the block x-axis by 15 degrees:

```
# t is a transformation matrix to rotate 15 degree around the x-axis
t = Matrix44.axis_rotate(axis=X_AXIS, angle=math.radians(15))
# transform block z-axis into new UCS z-axis (= extrusion vector)
uz = Vec3(t.transform(Z_AXIS))
# create new UCS at the insertion point, because we are rotating around the x-axis,
# ux is the same as the WCS x-axis and uz is the rotated z-axis.
ucs = UCS(origin=(1, 2, 0), ux=X_AXIS, uz=uz)
# transform insert location to OCS, block base_point=(0, 0, 0)
insert = ucs.to_ocs((0, 0, 0))
# for this case a rotation around the z-axis is not required
rotation = 0
```

(continues on next page)

(continued from previous page)

```
blockref = msp.add_blockref('CSYS', insert, dxftattribs={
    'extrusion': ucs.uz,
    'rotation': rotation,
})
```



The next example shows how to translate a block references with an already established OCS:

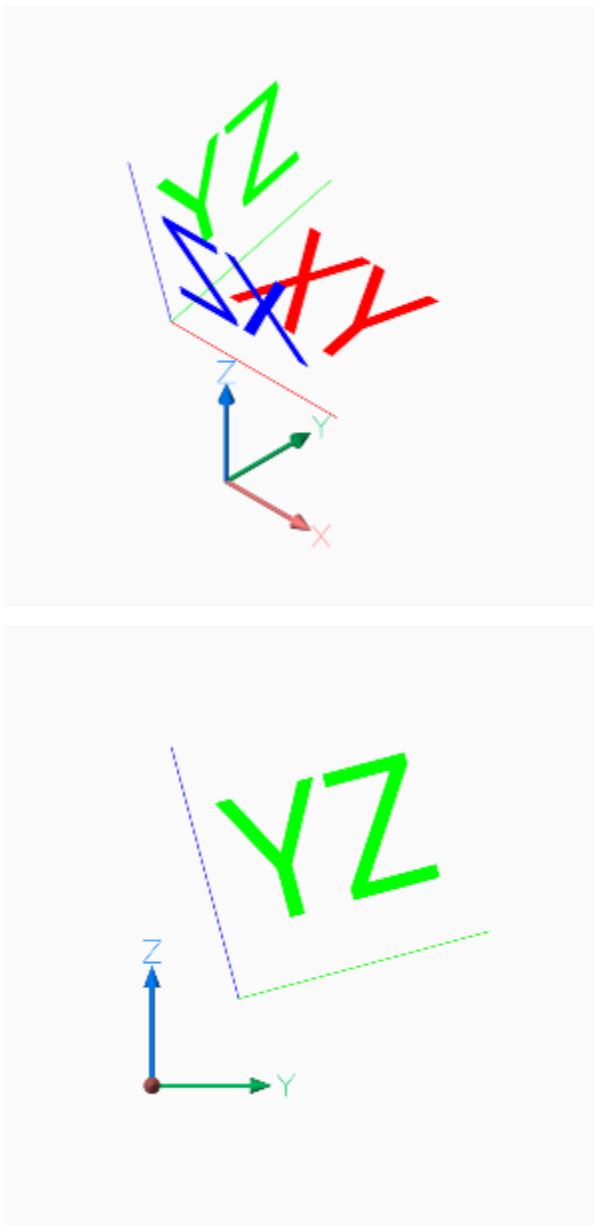
```
# translate a block references with an established OCS
translation = Vec3(-3, -1, 1)
# get established OCS
ocs = blockref.ocs()
# get insert location in WCS
actual_wcs_location = ocs.to_wcs(blockref.dxf.insert)
# translate location
```

(continues on next page)

(continued from previous page)

```
new_wcs_location = actual_wcs_location + translation
# convert WCS location to OCS location
blockref.dxf.insert = ocs.from_wcs(new_wcs_location)
```

Setting a new insert location is the same procedure without adding a translation vector, just transform the new insert location into the OCS.



The next operation is to rotate a block reference with an established OCS, rotation axis is the block y-axis, rotation angle is -90 degrees. First transform block y-axis (rotation axis) and block z-axis (extrusion vector) from OCS into WCS:

```
# rotate a block references with an established OCS around the block y-axis about 90_
_degree
ocs = blockref.ocs()
```

(continues on next page)

(continued from previous page)

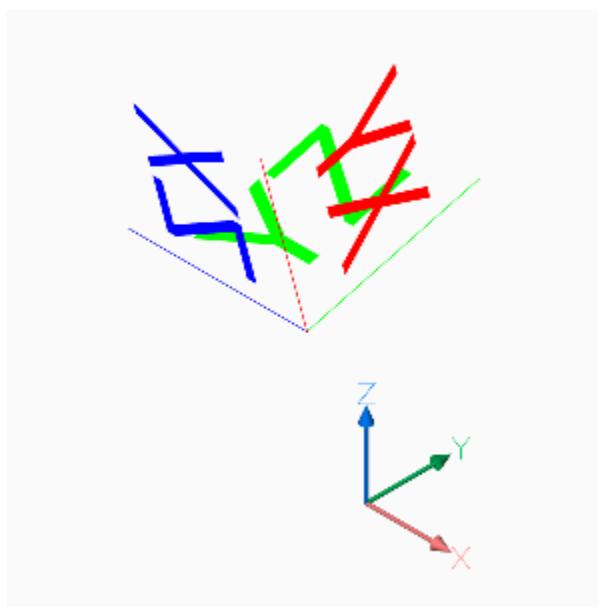
```
# convert block y-axis (= rotation axis) into WCS vector
rotation_axis = ocs.to_wcs((0, 1, 0))
# convert local z-axis (=extrusion vector) into WCS vector
local_z_axis = ocs.to_wcs((0, 0, 1))
```

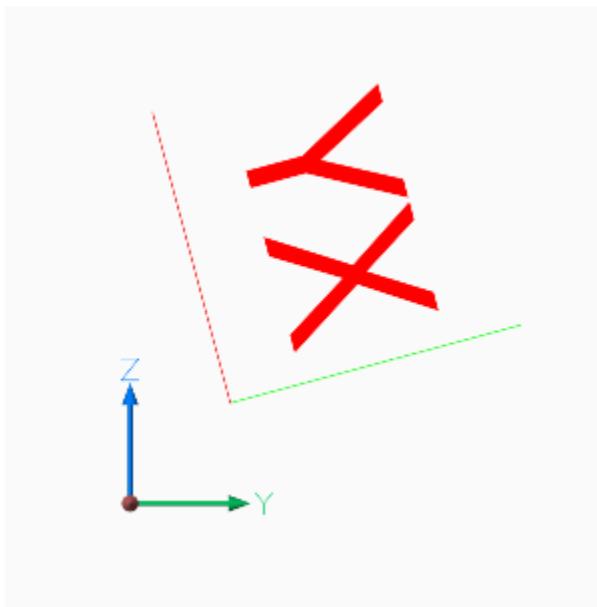
Build transformation matrix and transform extrusion vector and build new UCS:

```
# build transformation matrix
t = Matrix44.axis_rotate(axis=rotation_axis, angle=math.radians(-90))
uz = t.transform(local_z_axis)
uy = rotation_axis
# the block reference origin stays at the same location, no rotation needed
wcs_insert = ocs.to_wcs(blockref.dxf.insert)
# build new UCS to convert WCS locations and angles into OCS
ucs = UCS(origin=wcs_insert, uy=uy, uz=uz)
```

Set new OCS attributes, we also have to set the rotation attribute even though we do not rotate the block reference around the local z-axis, the new block x-axis (0 deg) differs from OCS x-axis and has to be adjusted:

```
# set new OCS
blockref.dxf.extrusion = ucs.uz
# set new insert
blockref.dxf.insert = ucs.to_ocs((0, 0, 0))
# set new rotation: we do not rotate the block reference around the local z-axis,
# but the new block x-axis (0 deg) differs from OCS x-axis and has to be adjusted
blockref.dxf.rotation = ucs.to_ocs_angle_deg(0)
```





And here is the point, where my math knowledge ends, for more advanced CAD operation you have to look elsewhere.

6.5.18 Tutorial for UCS Based Transformations

With *ezdxf* v0.11 a new feature for entity transformation was introduced, which makes working with OCS/UCS much easier, this is a new edition of the older *Tutorial for OCS/UCS Usage*. For the basic information read the old tutorial please. In *ezdxf* v0.13 the `transform_to_wcs()` interface was replaced by the general transformation interface: `transform()`.

For this tutorial we don't have to worry about the OCS and the extrusion vector, this is done automatically by the `transform()` method of each DXF entity.

Placing 2D Circle in 3D Space

To recreate the situation of the old tutorial instantiate a new UCS and rotate it around the local x-axis. Use UCS coordinates to place the 2D CIRCLE in 3D space, and transform the UCS coordinates to the WCS.

```
import math
import ezdxf
from ezdxf.math import UCS

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

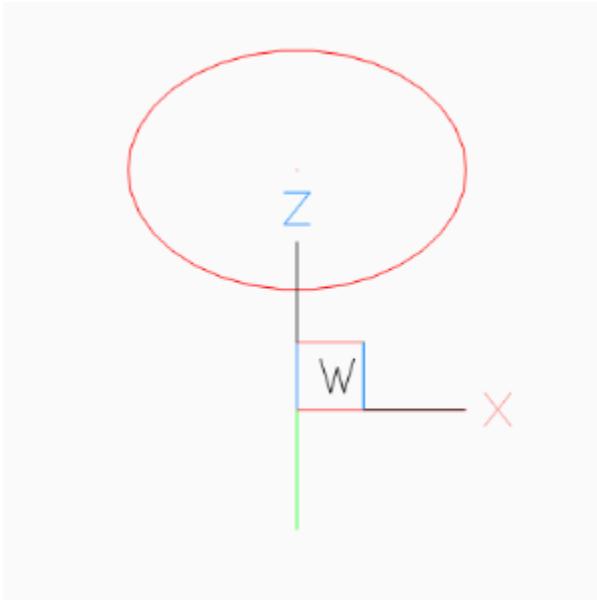
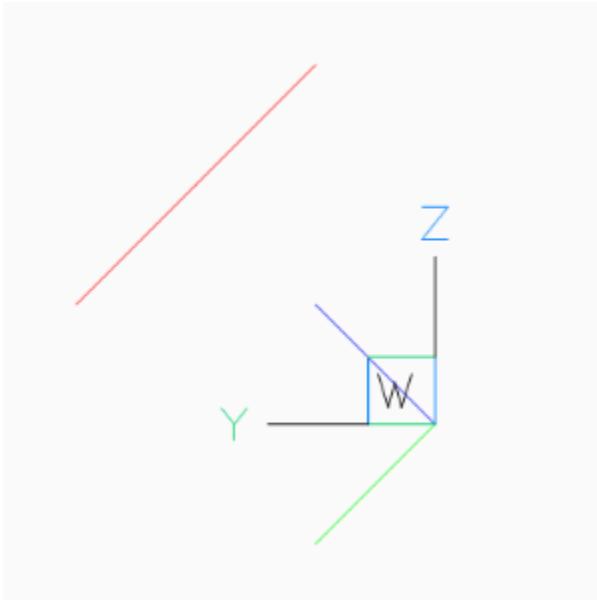
ucs = UCS() # New default UCS
# All rotation angles in radians, and rotation
# methods always return a new UCS.
ucs = ucs.rotate_local_x(math.radians(-45))
circle = msp.add_circle(
    # Use UCS coordinates to place the 2d circle in 3d space
    center=(0, 0, 2),
    radius=1,
    dxftattribs={'color': 1}
)
```

(continues on next page)

(continued from previous page)

```
circle.transform(ucs.matrix)

# mark center point of circle in WCS
msp.add_point((0, 0, 2), dxftattribs={'color': 1}).transform(ucs.matrix)
```



Placing LWPolyline in 3D Space

Simplified LPOLYLINE example:

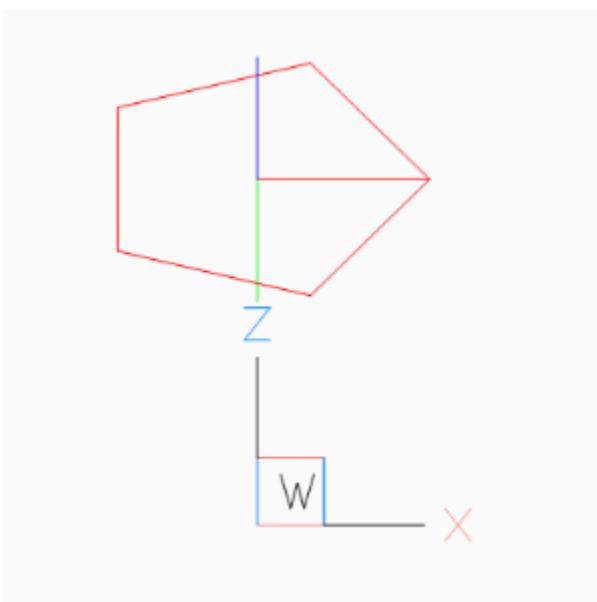
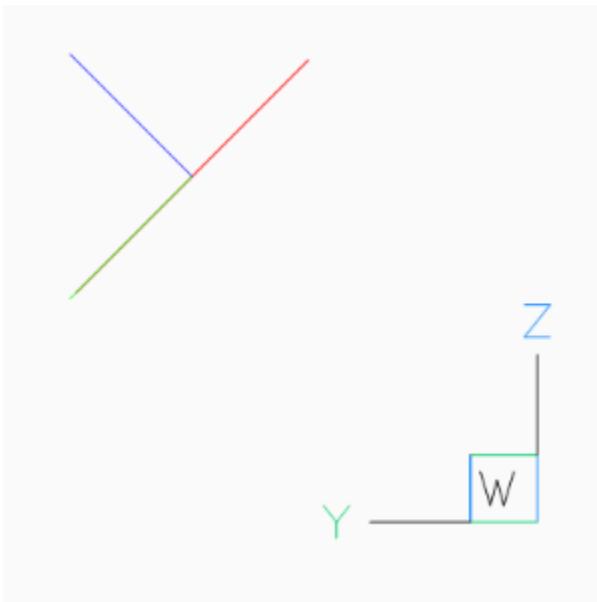
```
# The center of the pentagon should be (0, 2, 2), and the shape is
# rotated around x-axis about -45 degree
ucs = UCS(origin=(0, 2, 2)).rotate_local_x(math.radians(-45))
```

(continues on next page)

(continued from previous page)

```
msp.add_lwpolyline(  
    # calculating corner points in UCS coordinates  
    points=(Vec3.from_deg_angle((360 / 5) * n) for n in range(5)),  
    format='xy', # ignore z-axis  
    close=True,  
    dxftattribs={  
        'color': 1,  
    }
```

The 2D pentagon in 3D space in BricsCAD *Left* and *Front* view.



Using UCS to Place 3D Polyline

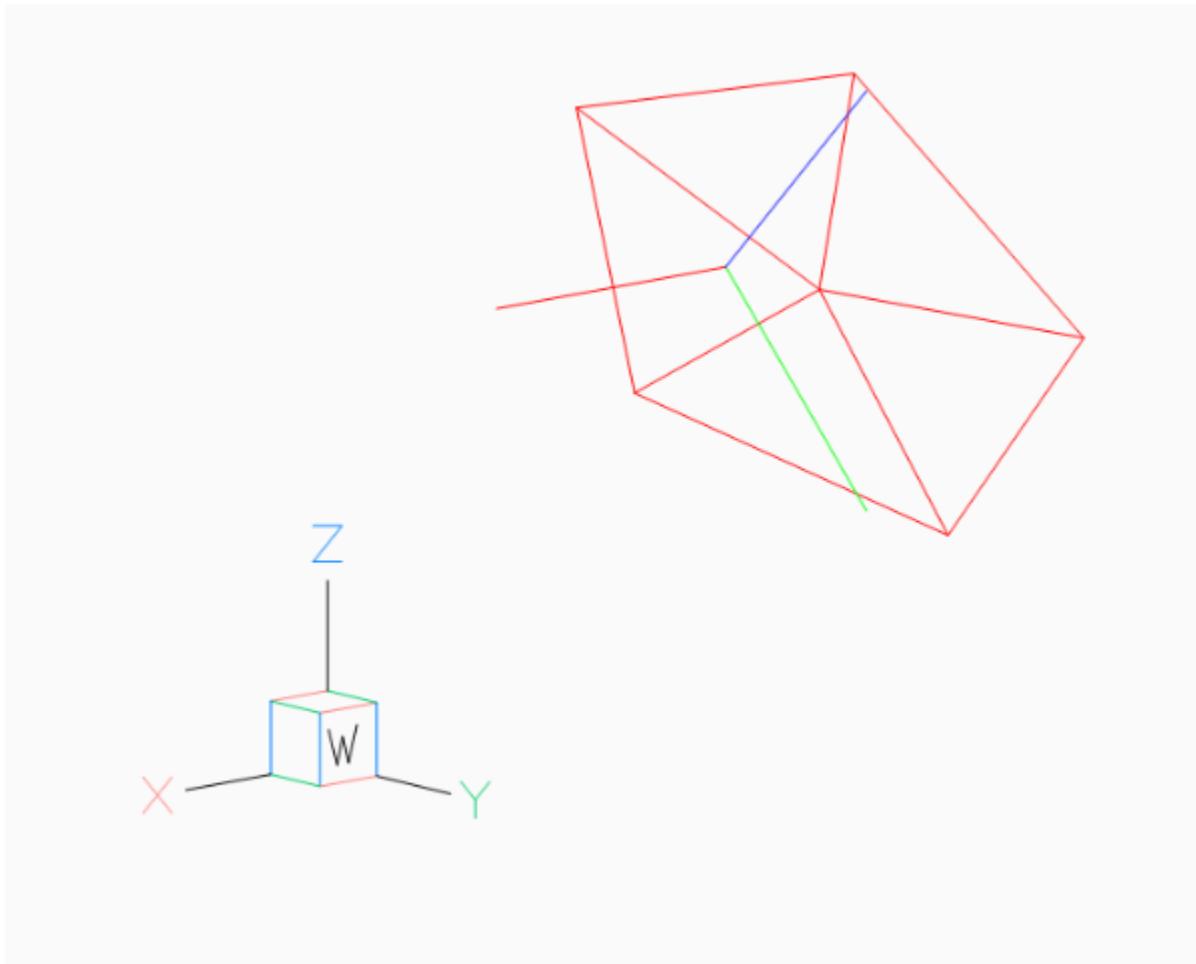
Simplified POLYLINE example: Using a first UCS to transform the POLYLINE and a second UCS to place the POLYLINE in 3D space.

```
# using an UCS simplifies 3D operations, but UCS definition can happen later
# calculating corner points in local (UCS) coordinates without Vec3 class
angle = math.radians(360 / 5)
corners_ucs = [(math.cos(angle * n), math.sin(angle * n), 0) for n in range(5)]

# let's do some transformations by UCS
transformation_ucs = UCS().rotate_local_z(math.radians(15)) # 1. rotation around z-
#axis
transformation_ucs.shift((0, .333, .333)) # 2. translation (inplace)
corners_ucs = list(transformation_ucs.points_to_wcs(corners_ucs))

location_ucs = UCS(origin=(0, 2, 2)).rotate_local_x(math.radians(-45))
msp.add_polyline3d(
    points=corners_ucs,
    close=True,
    dxftattribs={
        'color': 1,
    }
).transform(location_ucs.matrix)

# Add lines from the center of the POLYLINE to the corners
center_ucs = transformation_ucs.to_wcs((0, 0, 0))
for corner in corners_ucs:
    msp.add_line(
        center_ucs, corner, dxftattribs={'color': 1}
    ).transform(location_ucs.matrix)
```



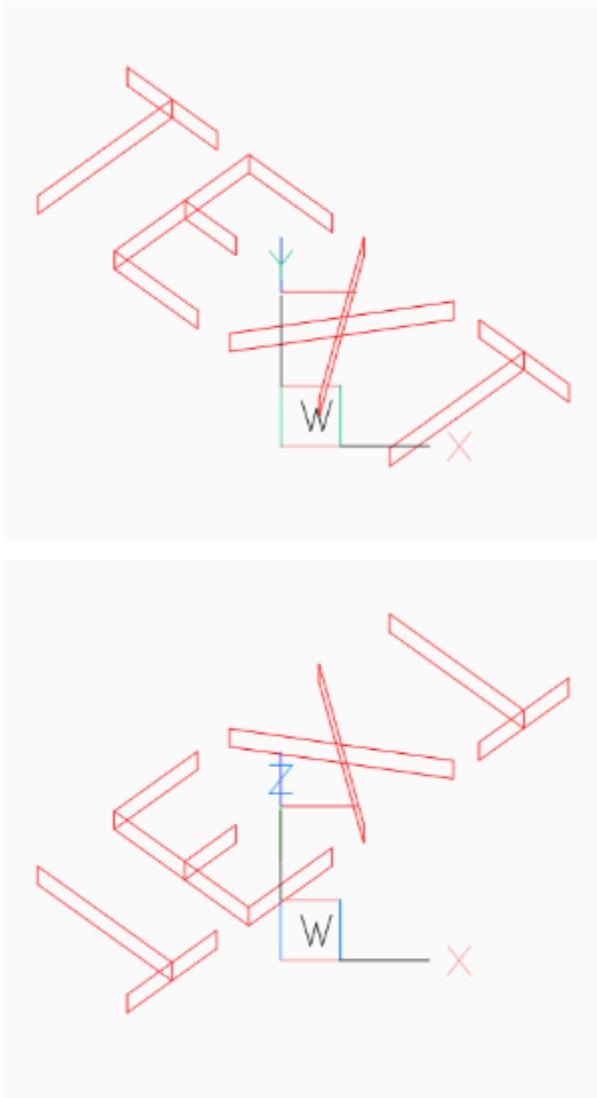
Placing 2D Text in 3D Space

The problem with the text rotation in the old tutorial disappears (or better it is hidden in `transform()`) with the new UCS based transformation method:

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

```
# thickness for text works only with shx fonts not with true type fonts
doc.styles.new('TXT', dxftattribs={'font': 'romans.shx'})

ucs = UCS(origin=(0, 2, 2)).rotate_local_x(math.radians(-45))
text = msp.add_text(
    text="TEXT",
    dxftattribs={
        # text rotation angle in degrees in UCS
        'rotation': -45,
        'thickness': .333,
        'color': 1,
        'style': 'TXT',
    }
)
# set text position in UCS
text.set_pos((0, 0, 0), align='MIDDLE_CENTER')
text.transform(ucs.matrix)
```



Placing 2D Arc in 3D Space

Same as for the text example, OCS angle transformation can be ignored:

```
ucs = UCS(origin=(0, 2, 2)).rotate_local_x(math.radians(-45))

CENTER = (0, 0)
START_ANGLE = 45
END_ANGLE = 270

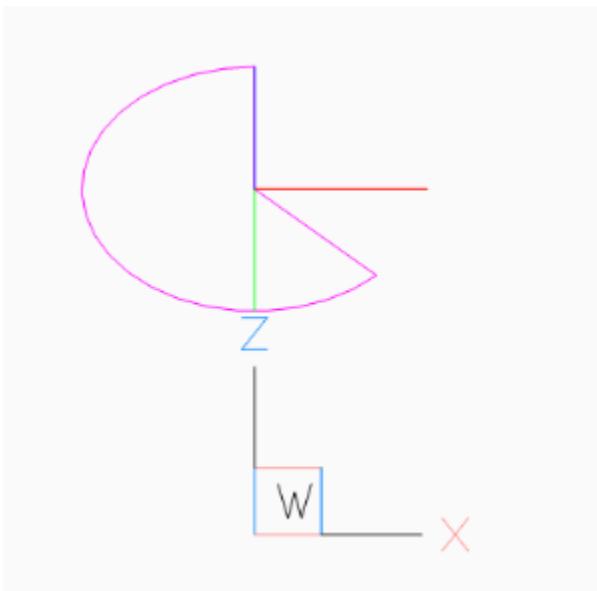
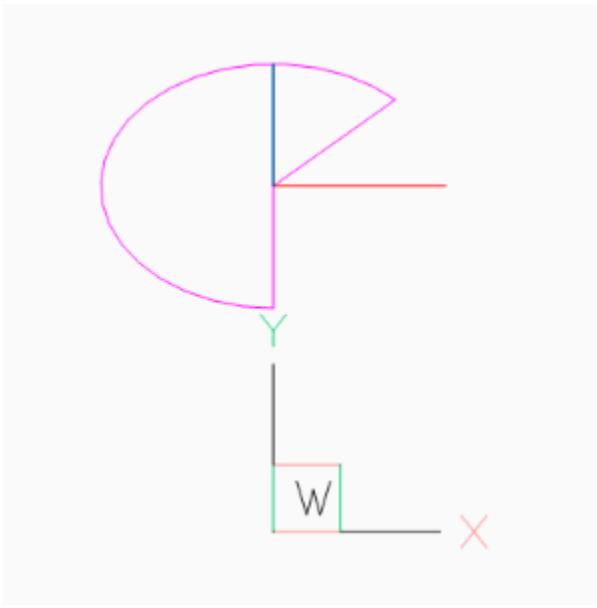
msp.add_arc(
    center=CENTER,
    radius=1,
    start_angle=START_ANGLE,
    end_angle=END_ANGLE,
    dxftattribs={'color': 6},
).transform(ucs.matrix)
```

(continues on next page)

(continued from previous page)

```
msp.add_line(
    start=CENTER,
    end=Vec3.from_deg_angle(START_ANGLE),
    dxftattribs={'color': 6},
).transform(ucs.matrix)

msp.add_line(
    start=CENTER,
    end=Vec3.from_deg_angle(END_ANGLE),
    dxftattribs={'color': 6},
).transform(ucs.matrix)
```



Placing Block References in 3D Space

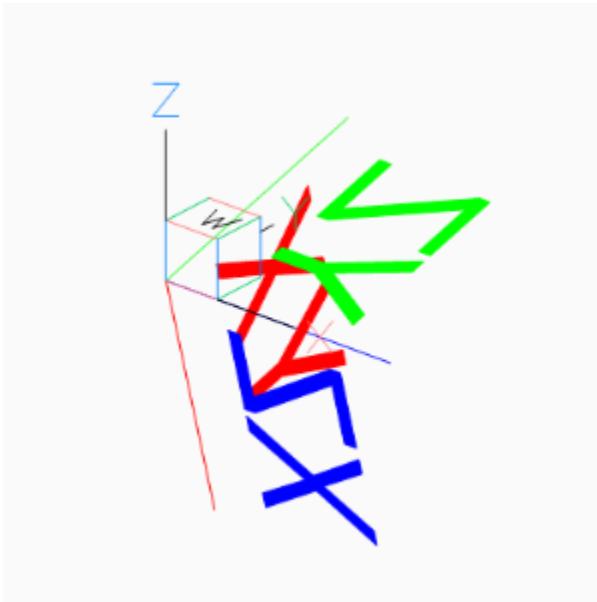
Despite the fact that block references (INSERT) can contain true 3D entities like LINE or MESH, the INSERT entity uses the same placing principle as TEXT or ARC shown in the previous chapters.

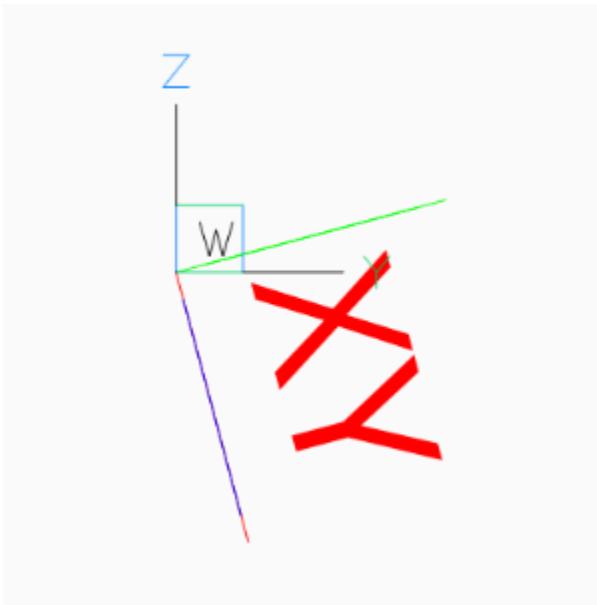
To rotate the block reference 15 degrees around the WCS x-axis, we place the block reference in the origin of the UCS, and rotate the UCS 90 degrees around its local y-axis, to align the UCS z-axis with the WCS x-axis:

This is just an excerpt of the important parts, see the whole code of [insert.py](#) at [github](#).

```
doc = ezdxf.new('R2010', setup=True)
blk = doc.blocks.new('CSYS')
setup_csys(blk)
msp = doc.modelspace()

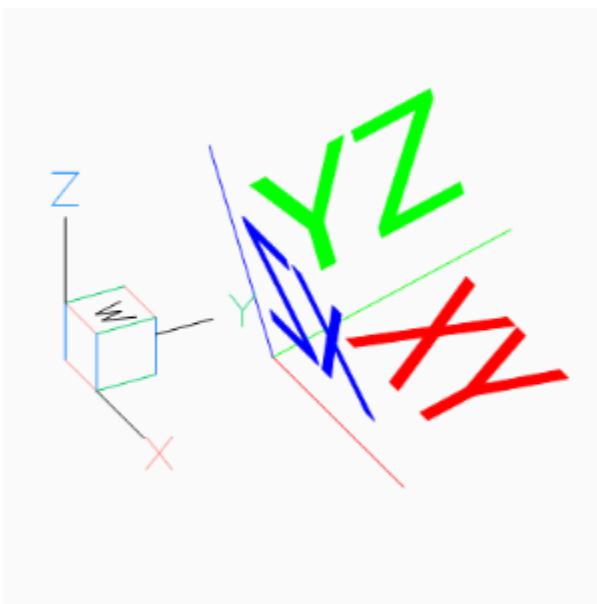
ucs = UCS().rotate_local_y(angle=math.radians(90))
msp.add_blockref(
    'CSYS',
    insert=(0, 0),
    # rotation around the block z-axis (= WCS x-axis)
    dxftattribs={'rotation': 15},
).transform(ucs.matrix)
```

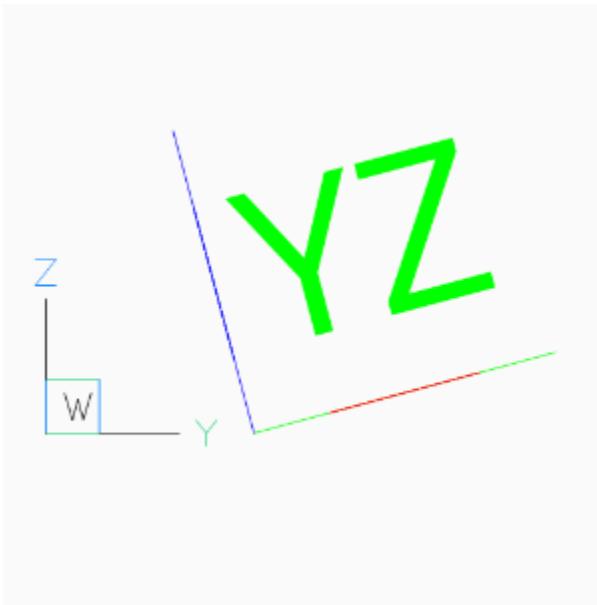




A more simple approach is to ignore the `rotate` attribute at all and just rotate the UCS. To rotate a block reference around any axis rather than the block z-axis, rotate the UCS into the desired position. Following example rotates the block reference around the block x-axis by 15 degrees:

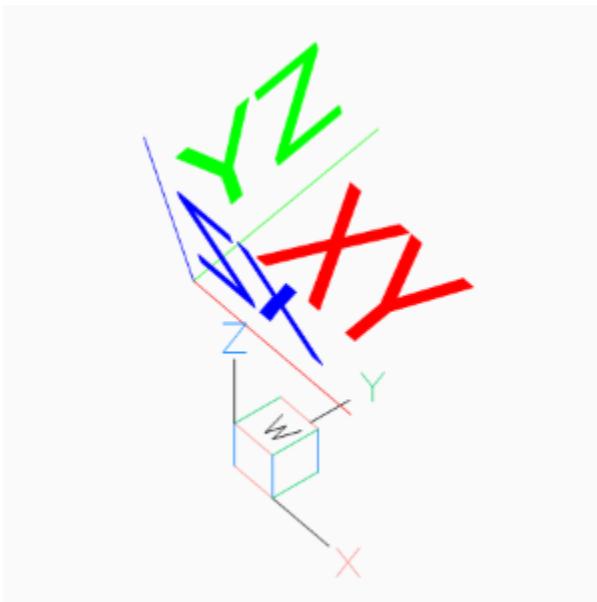
```
ucs = UCS(origin=(1, 2, 0)).rotate_local_x(math.radians(15))
blockref = msp.add_blockref('CSYS', insert=(0, 0, 0))
blockref.transform(ucs.matrix)
```





The next example shows how to translate a block references with an already established OCS:

```
# New UCS at the translated location, axis aligned to the WCS
ucs = UCS((-3, -1, 1))
# Transform an already placed block reference, including
# the transformation of the established OCS.
blockref.transform(ucs.matrix)
```





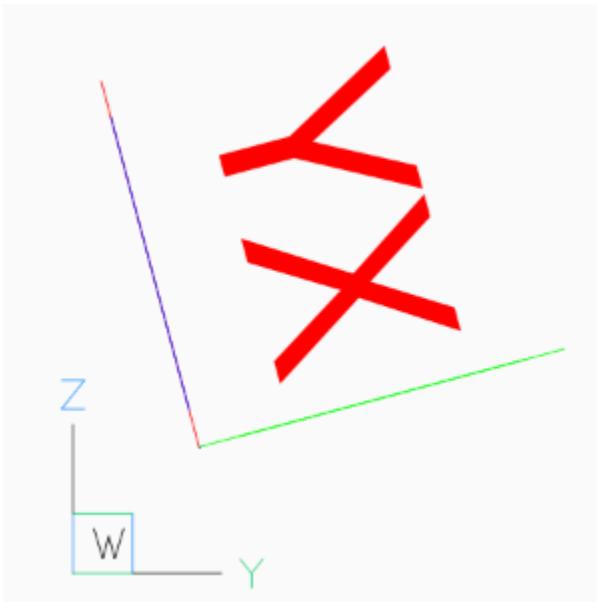
The next operation is to rotate a block reference with an established OCS, rotation axis is the block y-axis, rotation angle is -90 degrees. The idea is to create an UCS in the origin of the already placed block reference, UCS axis aligned to the block axis and resetting the block reference parameters for a new WCS transformation.

```
# Get UCS at the block reference insert location, UCS axis aligned
# to the block axis.
ucs = blockref.ucs()
# Rotate UCS around the local y-axis.
ucs = ucs.rotate_local_y(math.radians(-90))
```

Reset block reference parameters, this places the block reference in the UCS origin and aligns the block axis to the UCS axis, now we do a new transformation from UCS to WCS:

```
# Reset block reference parameters to place block reference in
# UCS origin, without any rotation and OCS.
blockref.reset_transformation()

# Transform block reference from UCS to WCS
blockref.transform(ucs.matrix)
```



6.5.19 Tutorial for Linear Dimensions

The [Dimension](#) entity is the generic entity for all dimension types, but unfortunately AutoCAD is **not willing** to show a dimension line defined only by this dimension entity, it also needs an anonymous block which contains the dimension line *rendering* in this documentation, beside the fact this is not a real graphical rendering. BricsCAD is a much more friendly CAD application, which do show the dimension entity without the graphical rendering as block, which was very useful for testing, because there is no documentation how to apply all the dimension style variables (more than 80). This seems to be the reason why dimension lines are rendered so differently by many CAD application.

Don't expect to get the same rendering results by *ezdxf* as you get from AutoCAD, *ezdxf* tries to be as close to the results rendered by BricsCAD, but it is not possible to implement all the various combinations of dimension style parameters, which often affect one another.

Note: *Ezdx*f does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Text rendering is another problem, because *ezdxf* has no real rendering engine. Some font properties, like the real text width, which is only available to *ezdxf* if the *Matplotlib* package is installed and may also vary slightly for different CAD applications. Without access to the *Matplotlib* package the text properties in *ezdxf* are based on an abstract monospaced font and are bigger than required by true type fonts.

Not all DIMENSION and DIMSTYLE features are supported by all DXF versions, especially DXF R12 does not support many features, but in this case the required rendering of dimension lines is an advantage, because if the application just shows the rendered block, all features which can be used in DXF R12 are displayed like linetypes, but this features will disappear if the dimension line will be edited in the CAD application. *Ezdx*f writes only the supported DIMVARS of the used DXF version to avoid invalid DXF files. So it is not that critical to know all the supported features of a DXF version, except for limits and tolerances, *ezdxf* uses the advanced features of the MTEXT entity to create limits and tolerances and therefore they are not supported (displayed) in DXF R12 files.

See also:

- Graphical reference of many DIMVARS and some advanced information: [DIMSTYLE Table](#)
- Source code file `standards.py` shows how to create your own DIMSTYLES.
- The Script `dimension_linear.py` shows examples for linear dimensions.

Horizontal Dimension

```
import ezdxf

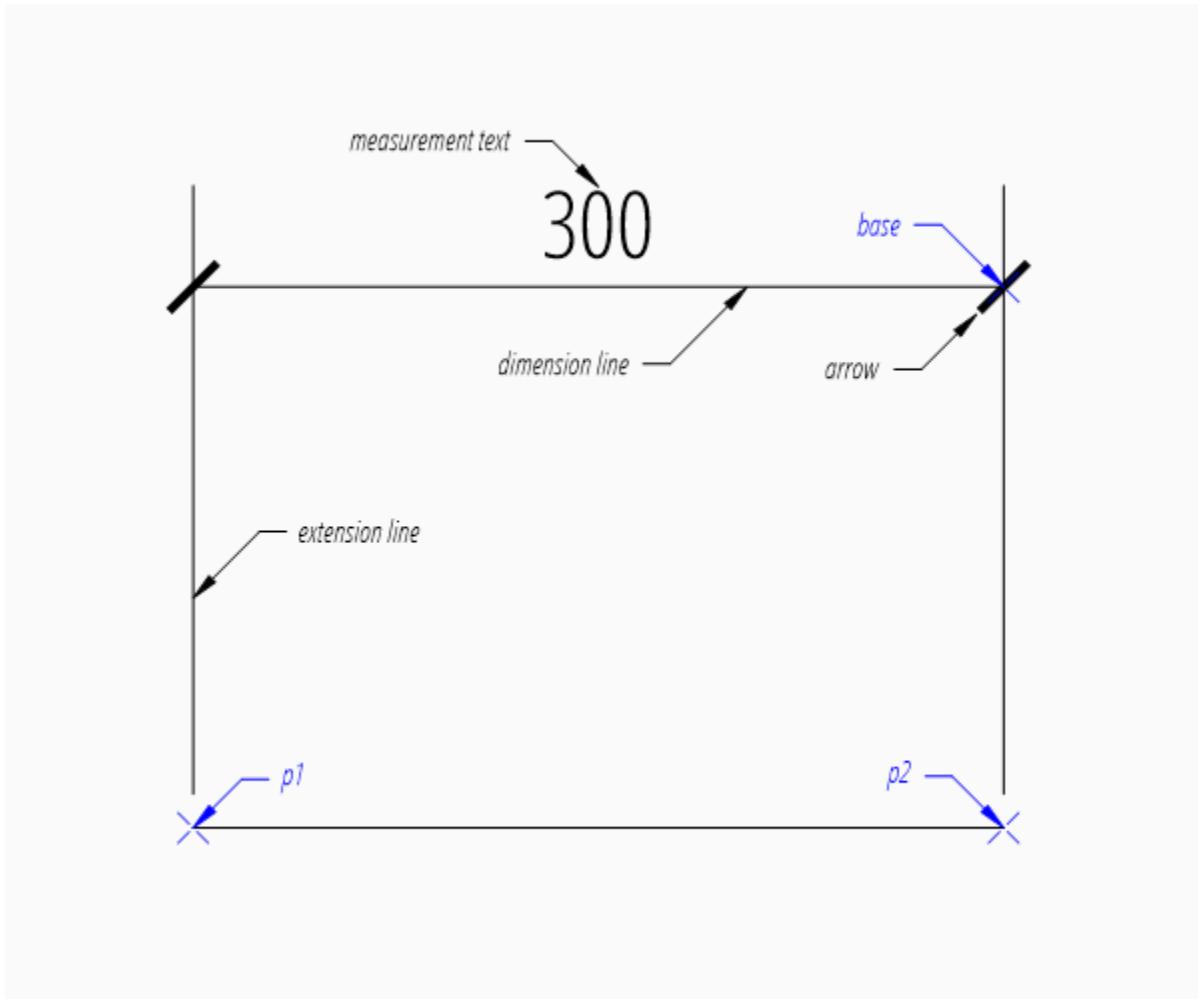
# Create a DXF R2010 document:
# Use argument setup=True to setup the default dimension styles.
doc = ezdxf.new("R2010", setup=True)

# Add new dimension entities to the modelspace:
msp = doc.modelspace()

# Add a LINE entity for visualization, not required to create the DIMENSION
# entity:
msp.add_line((0, 0), (3, 0))

# Add a horizontal linear DIMENSION entity:
dim = msp.add_linear_dim(
    base=(3, 2), # location of the dimension line
    p1=(0, 0), # 1st measurement point
    p2=(3, 0), # 2nd measurement point
    dimstyle="EZDXF", # default dimension style
)

# Necessary second step to create the BLOCK entity with the dimension geometry.
# Additional processing of the DIMENSION entity could happen between adding
# the entity and the rendering call.
dim.render()
doc.saveas("dim_linear_horiz.dxf")
```



The example above creates a horizontal `Dimension` entity. The default dimension style “EZDXF” is defined as:

- 1 drawing unit = 1m
- measurement text height = 0.25 (drawing scale = 1:100)
- the length factor `dimlfac` = 100, which creates a measurement text in cm.
- arrow is “ARCHTICK”, arrow size `dimasz` = 0.175

Every dimension style which does not exist will be replaced by the dimension style “Standard” at DXF export by `save()` or `saveas()` (e.g. dimension style setup was not initiated).

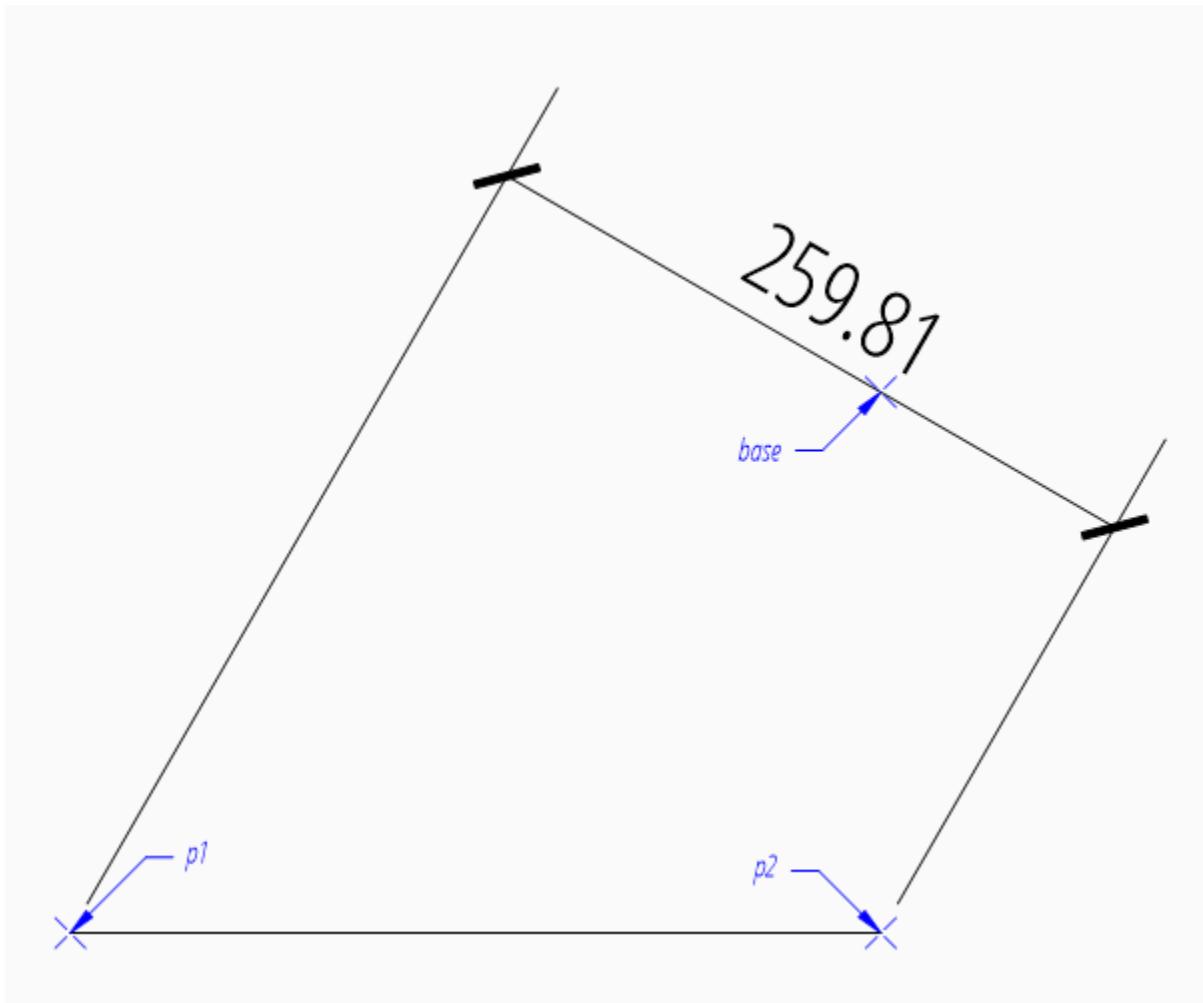
The `base` point defines the location of the dimension line, `ezdxf` accepts any point on the dimension line, the point `p1` defines the start point of the first extension line, which also defines the first measurement point and the point `p2` defines the start point of the second extension line, which also defines the second measurement point.

The return value `dim` is **not** a dimension entity, instead a `DimStyleOverride` object is returned, the dimension entity is stored as attribute `dim.dimension`.

Vertical and Rotated Dimension

Argument `angle` defines the angle of the dimension line in relation to the x-axis of the WCS or UCS, measurement is the distance between first and second measurement point in direction of `angle`.

```
# assignment to dim is not necessary, if no additional processing happens
msp.add_linear_dim(base=(3, 2), p1=(0, 0), p2=(3, 0), angle=-30).render()
doc.saveas("dim_linear_rotated.dxf")
```

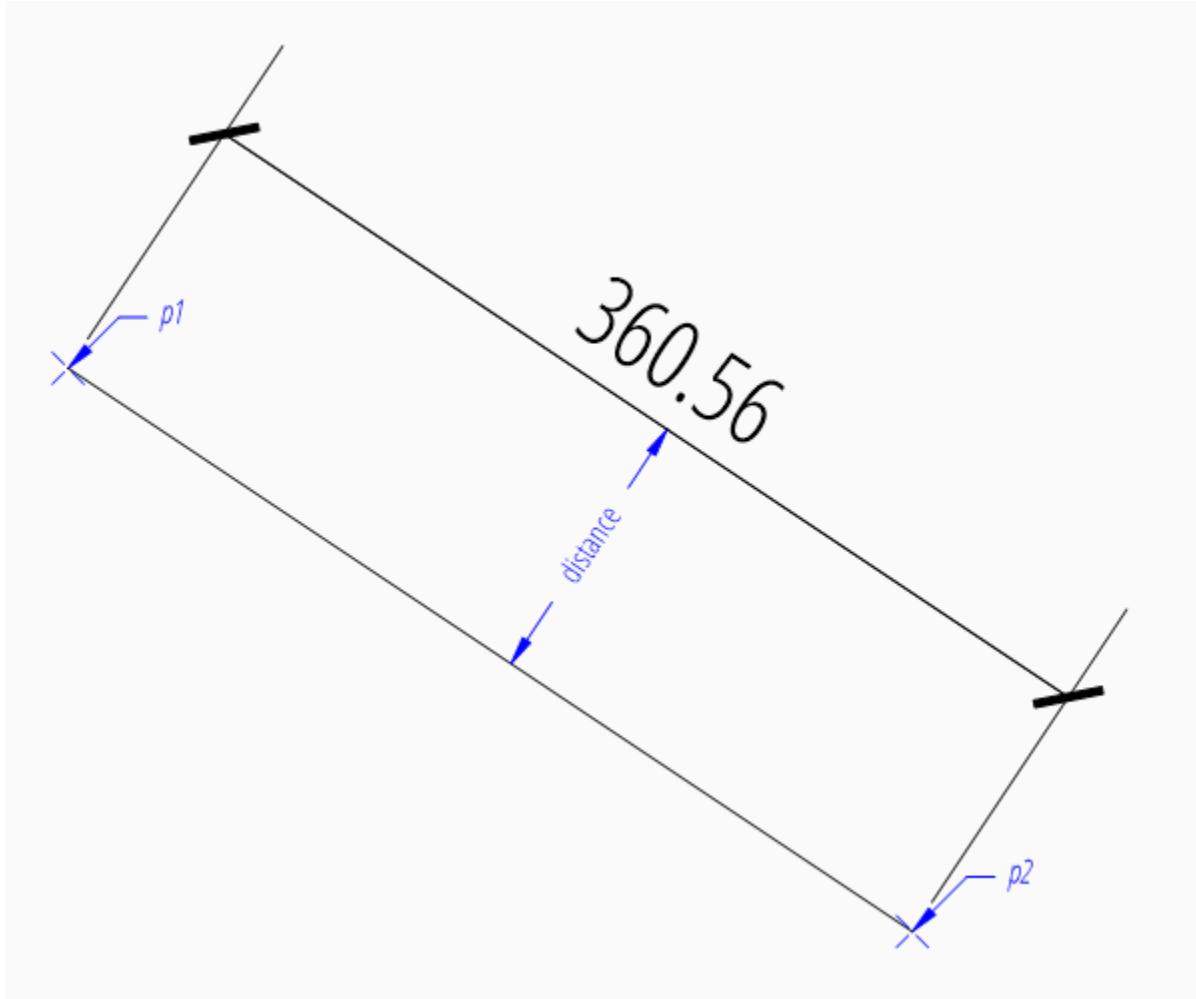


For a vertical dimension set argument *angle* to 90 degree, but in this example the vertical distance would be 0.

Aligned Dimension

An aligned dimension line is parallel to the line defined by the definition points *p1* and *p2*. The placement of the dimension line is defined by the argument *distance*, which is the distance between the definition line and the dimension line. The *distance* of the dimension line is orthogonal to the base line in counter clockwise orientation.

```
msp.add_line((0, 2), (3, 0))
dim = msp.add_aligned_dim(p1=(0, 2), p2=(3, 0), distance=1)
doc.saveas("dim_linear_aligned.dxf")
```



Dimension Style Override

Many dimension styling options are defined by the associated `DimStyle` entity. But often you wanna change just a few settings without creating a new dimension style, therefore the DXF format has a protocol to store this changed settings in the dimension entity itself. This protocol is supported by `ezdxf` and every factory function which creates dimension entities supports the `override` argument. This `override` argument is a simple Python dictionary (e.g. `override = {"dimtad": 4}`, place measurement text below dimension line).

The overriding protocol is managed by the `DimStyleOverride` object, which is returned by the most dimension factory functions.

Placing Measurement Text

The default location of the measurement text depends on various `DimStyle` parameters and is applied if no user defined text location is defined.

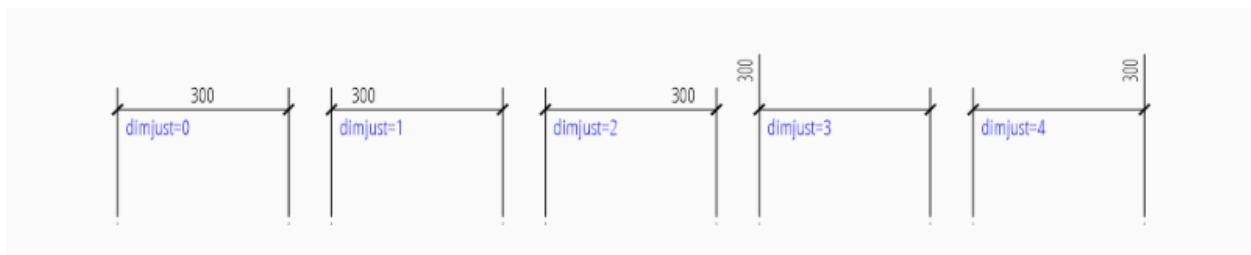
Default Text Locations

“Horizontal direction” means in direction of the dimension line and “vertical direction” means perpendicular to the dimension line direction.

The “horizontal” location of the measurement text is defined by `dimjust`:

0	Center of dimension line
1	Left side of the dimension line, near first extension line
2	Right side of the dimension line, near second extension line
3	Over first extension line
4	Over second extension line

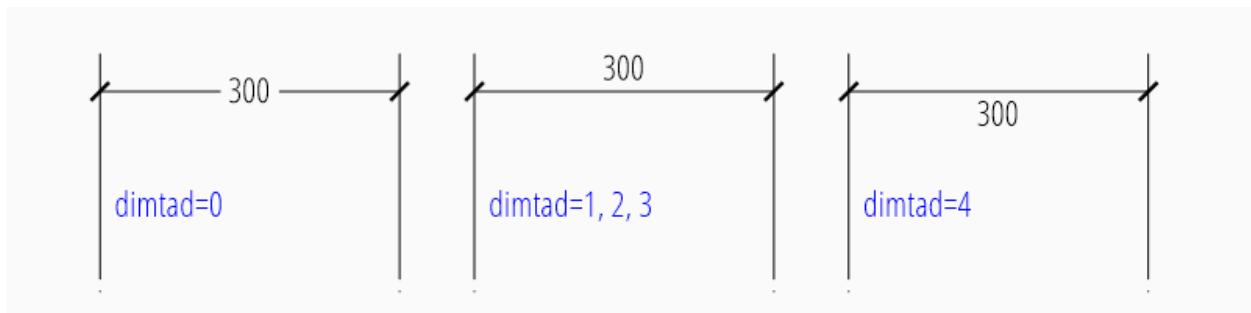
```
msp.add_linear_dim(
    base=(3, 2), p1=(0, 0), p2=(3, 0), override={"dimjust": 1}
).render()
```



The “vertical” location of the measurement text relative to the dimension line is defined by `dimtad`:

0	Center, it is possible to adjust the vertical location by <code>dimtvp</code>
1	Above
2	Outside, handled like <code>Above</code> by <code>ezdxf</code>
3	JIS, handled like <code>Above</code> by <code>ezdxf</code>
4	Below

```
msp.add_linear_dim(
    base=(3, 2), p1=(0, 0), p2=(3, 0), override={"dimtad": 4}
).render()
```



The distance between text and dimension line is defined by `dimgap`.

The `DimStyleOverride` object has a method `set_text_align()` to set the default text location in an easy way, this is also the reason for the 2 step creation process of dimension entities:

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

halign	“left”, “right”, “center”, “above1”, “above2”
valign	“above”, “center”, “below”

Run function `example_for_all_text_placings_R2007()` in the example script `dimension_linear.py` to create a DXF file with all text placings supported by `ezdxf`.

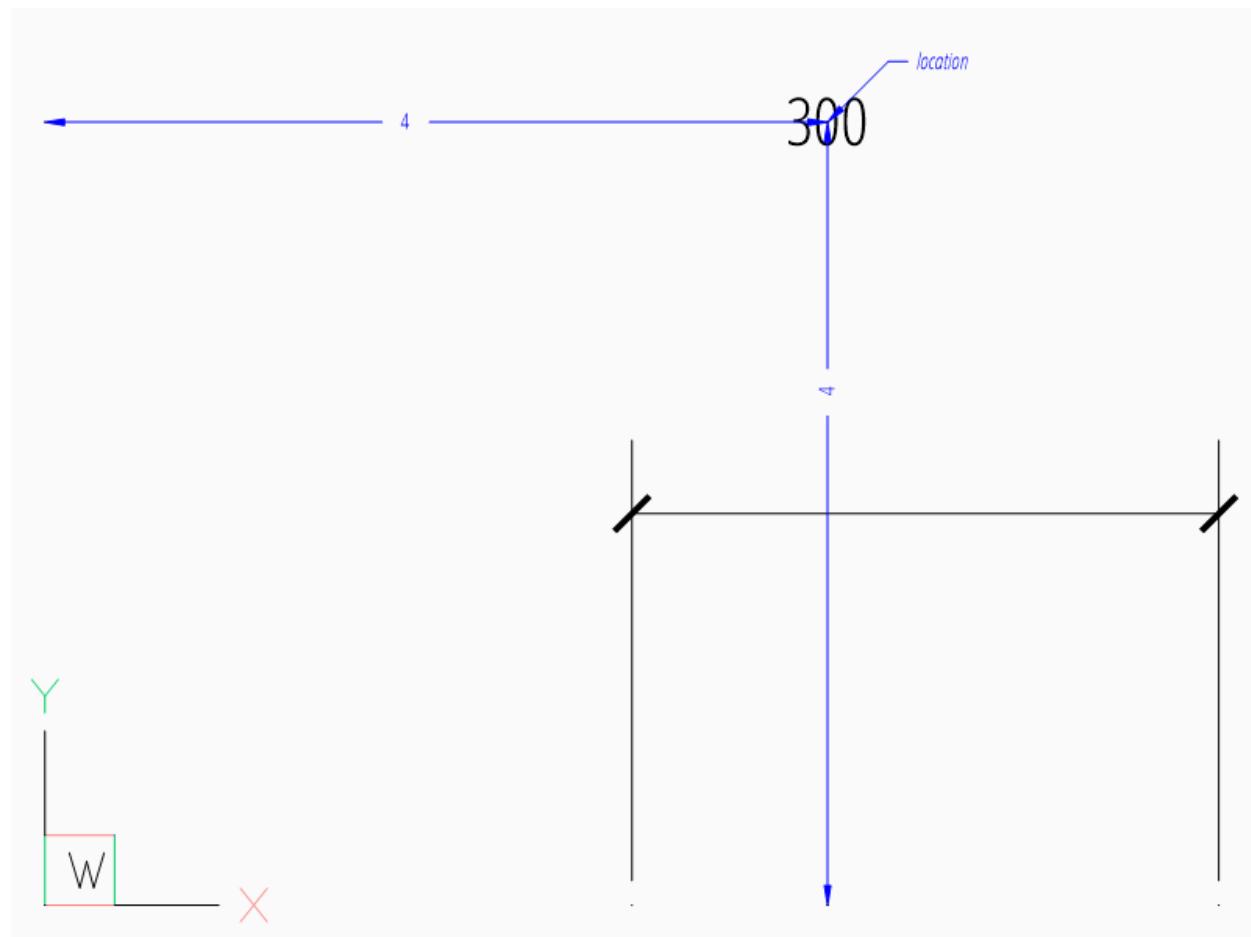
User Defined Text Locations

Beside the default location, it is possible to locate the measurement text freely.

Location Relative to Origin

The user defined text location can be set by the argument `location` in most dimension factory functions and always references the midpoint of the measurement text:

```
msp.add_linear_dim(  
    base=(3, 2), p1=(3, 0), p2=(6, 0), location=(4, 4)  
)
```

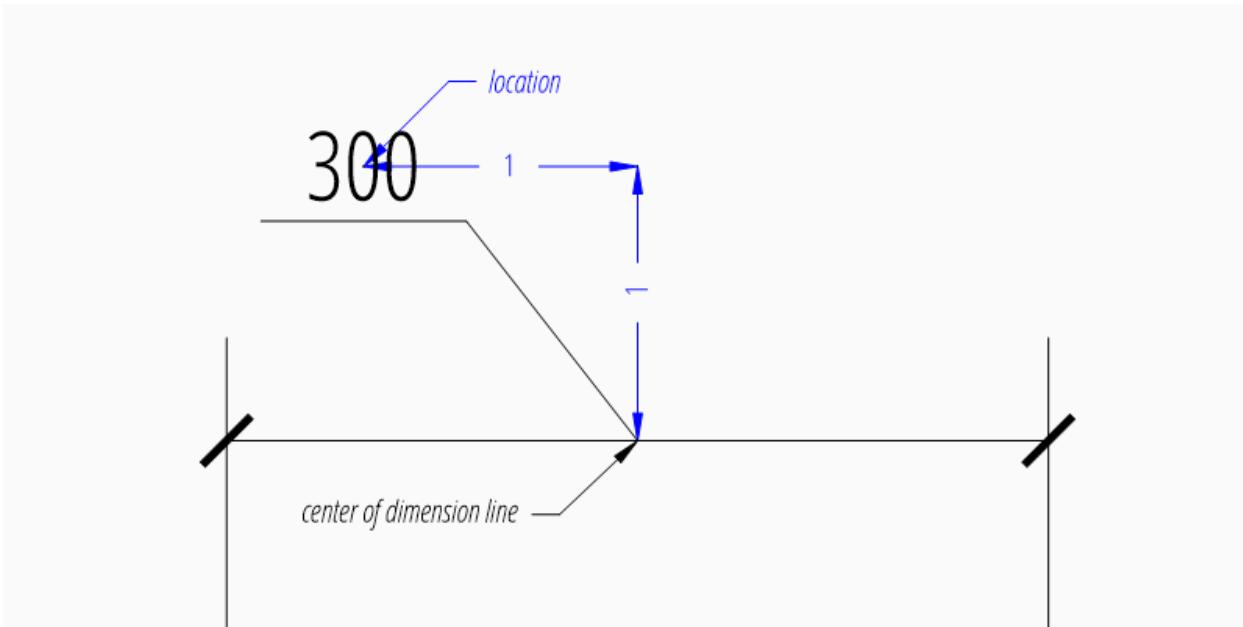


The `location` is relative to origin of the active coordinate system or WCS if no UCS is defined in the `render()` method, the user defined `location` can also be set by `user_location_override()`.

Location Relative to Center of Dimension Line

The method `set_location()` has additional features for linear dimensions. Argument `leader = True` adds a simple leader from the measurement text to the center of the dimension line and argument `relative = True` places the measurement text relative to the center of the dimension line.

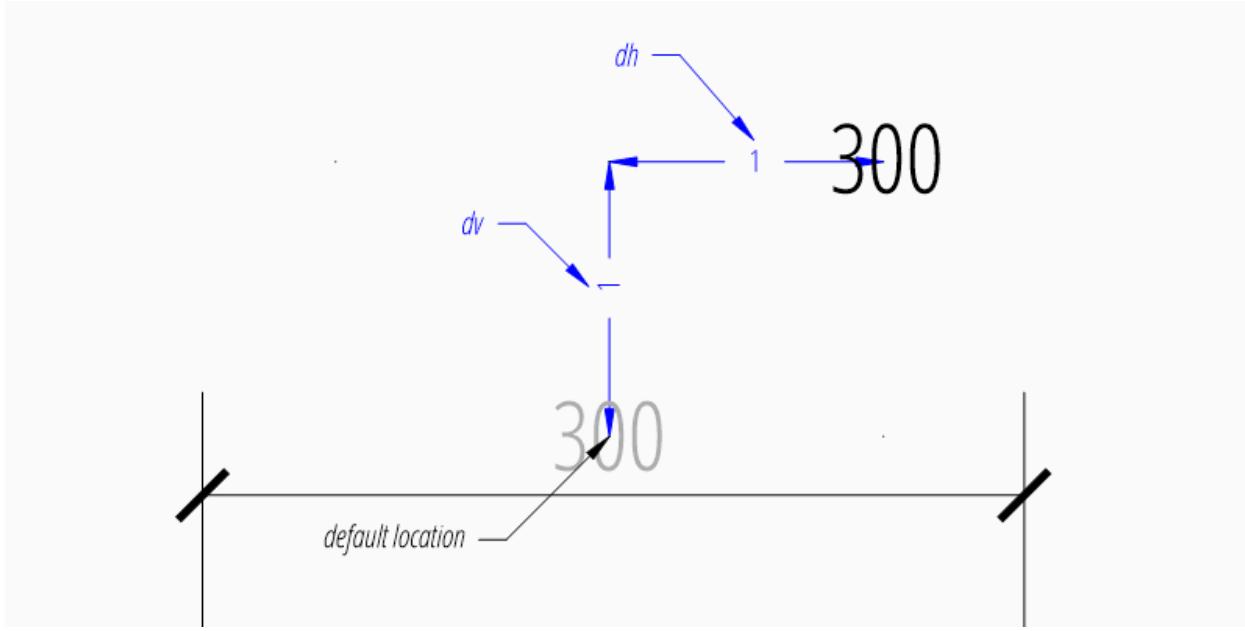
```
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_location(location=(-1, 1), leader=True, relative=True)
dim.render()
```



Location Relative to Default Location

The method `shift_text()` shifts the measurement text away from the default text location. The shifting directions are aligned to the text direction, which is the direction of the dimension line in most cases, `dh` (for delta horizontal) shifts the text parallel to the text direction, `dv` (for delta vertical) shifts the text perpendicular to the text direction. This method does not support leaders.

```
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.shift_text(dh=1, dv=1)
dim.render()
```



Overriding Text Rotation

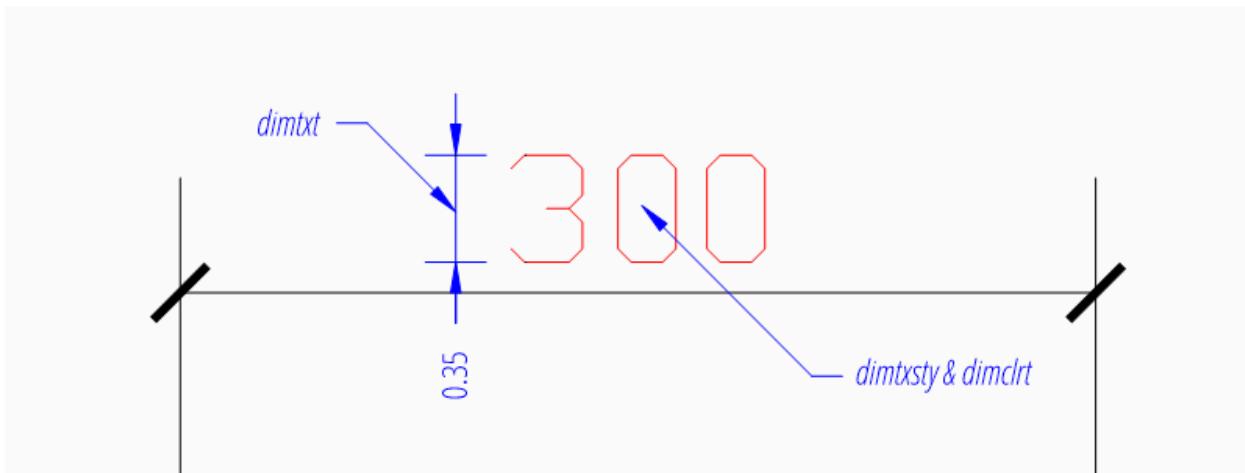
All factory methods supporting the argument `text_rotation` can override the measurement text rotation. The user defined rotation is relative to the render UCS x-axis (default is WCS).

Measurement Text Formatting and Styling

Text Properties

DIMVAR	Description
<code>dimtxsty</code>	Specifies the text style of the dimension as <code>Textstyle</code> name.
<code>dimtxt</code>	Text height in drawing units.
<code>dimclrt</code>	Measurement text color as <code>AutoCAD Color Index (ACI)</code> .

```
msp.add_linear_dim(
    base=(3, 2),
    p1=(3, 0),
    p2=(6, 0),
    override={
        "dimtxsty": "Standard",
        "dimtxt": 0.35,
        "dimclrt": 1,
    }
).render()
```



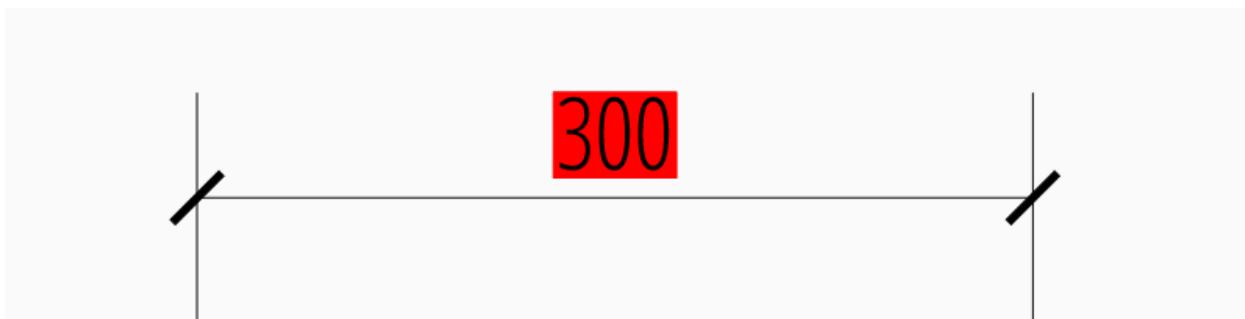
Background Filling

Background fillings are supported since DXF R2007, and *ezdxf* uses the MTEXT entity to implement this feature, so setting background filling in DXF R12 has no effect. The DIMVAR `dimtfill` defines the kind of background filling and the DIMVAR `dimtfillclr` defines the fill color.

DIMVAR	Description
<code>dimtfill</code>	Enables background filling if bigger than 0
<code>dimtfillclr</code>	Fill color as <i>AutoCAD Color Index (ACI)</i> , if <code>dimtfill</code> is 2

dimtfill	Description
0	disabled
1	canvas color
2	color defined by <code>dimtfillclr</code>

```
msp.add_linear_dim(
    base=(3, 2),
    p1=(3, 0),
    p2=(6, 0),
    override={
        "dimtfill": 2,
        "dimtfillclr": 1,
    }
).render()
```



Text Formatting

- **decimal places:** `dimdec` defines the number of decimal places displayed for the primary units of a dimension. (DXF R2000)
- **decimal point character:** `dimdsep` defines the decimal point as ASCII code, get the ASCII code by `ord('.')`
- **rounding:** `dimrnd`, rounds all dimensioning distances to the specified value, for instance, if `dimrnd` is set to 0.25, all distances round to the nearest 0.25 unit. If `dimrnd` is set to 1.0, all distances round to the nearest integer. For more information look at the documentation of the `ezdxf.math.xround()` function.
- **zero trimming:** `dimzin`, `ezdxf` supports only a subset of values:
 - 4 to suppress leading zeros
 - 8 to suppress trailing zeros
 - 12 as the combination of both
- **measurement factor:** scale measurement by factor `dimlfac`, e.g. to get the dimensioning text in cm for a DXF file where 1 drawing unit represents 1m, set `dimlfac` to 100.
- **text template:** `dimpost`, “<>” represents the measurement text, e.g. “~<>cm” produces “~300cm” for measurement in previous example.

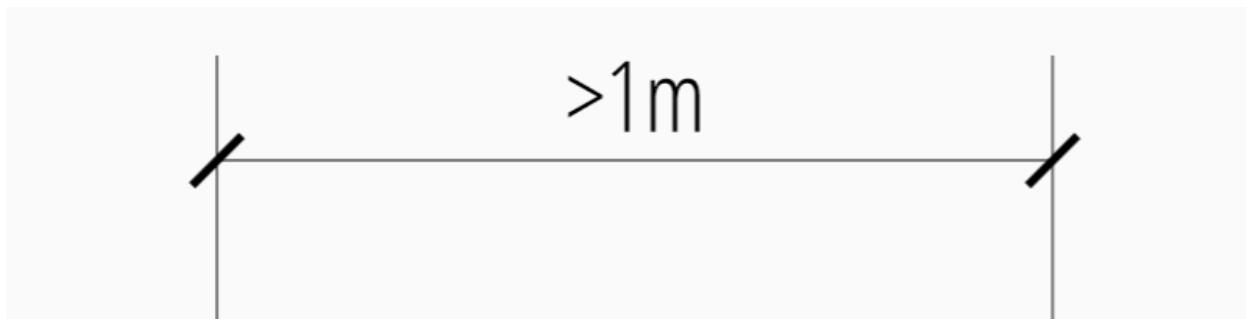
To set this values the `ezdxf.entities.DimStyle.set_text_format()` and `ezdxf.entities.DimStyleOverride.set_text_format()` methods are very recommended.

Overriding Measurement Text

This feature allows overriding the real measurement text by a custom measurement text, the text is stored as string in the `Dimension` entity as attribute `text`. Special values of the `text` attribute are: one space “ ” to suppress the measurement text at all, an empty string “” or “<>” to display the real measurement.

All factory functions have an explicit `text` argument, which always replaces the `text` value in the `dxftattribs` dict.

```
msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0), text=">1m").render()
```

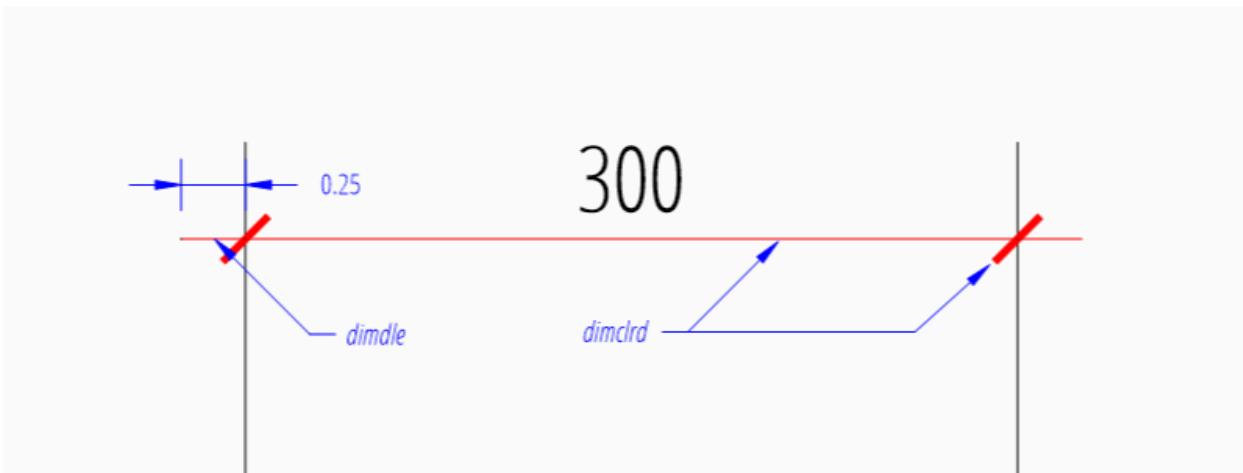


Dimension Line Properties

The *dimension line color* is defined by the DIMVAR `dimclrd` as *AutoCAD Color Index (ACI)*, `dimclrd` and also defines the color of the arrows. The *linetype* is defined by `dimltype` and requires DXF R2007. The *linewidth* is defined by `dimlwd` and requires DXF R2000, see also the `lineweight` reference for valid values. The `dimdle` is the extension of the dimension line beyond the extension lines, this dimension line extension is not supported for all arrows.

DIMVAR	Description
dimclrd	dimension line and arrows color as <i>AutoCAD Color Index (ACI)</i>
dimltype	linetype of dimension line
dimlwd	line weight of dimension line
dimdle	extension of dimension line in drawing units

```
msp.add_linear_dim(
    base=(3, 2),
    p1=(3, 0),
    p2=(6, 0),
    override={
        "dimclrd": 1, # red
        "dimdle": 0.25,
        "dimltype": "DASHED2",
        "dimlwd": 35, # 0.35mm line weight
    }
).render()
```



`DimStyleOverride()` method:

```
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_dimline_format(
    color=1, linetype="DASHED2", linewidth=35, extension=0.25
)
dim.render()
```

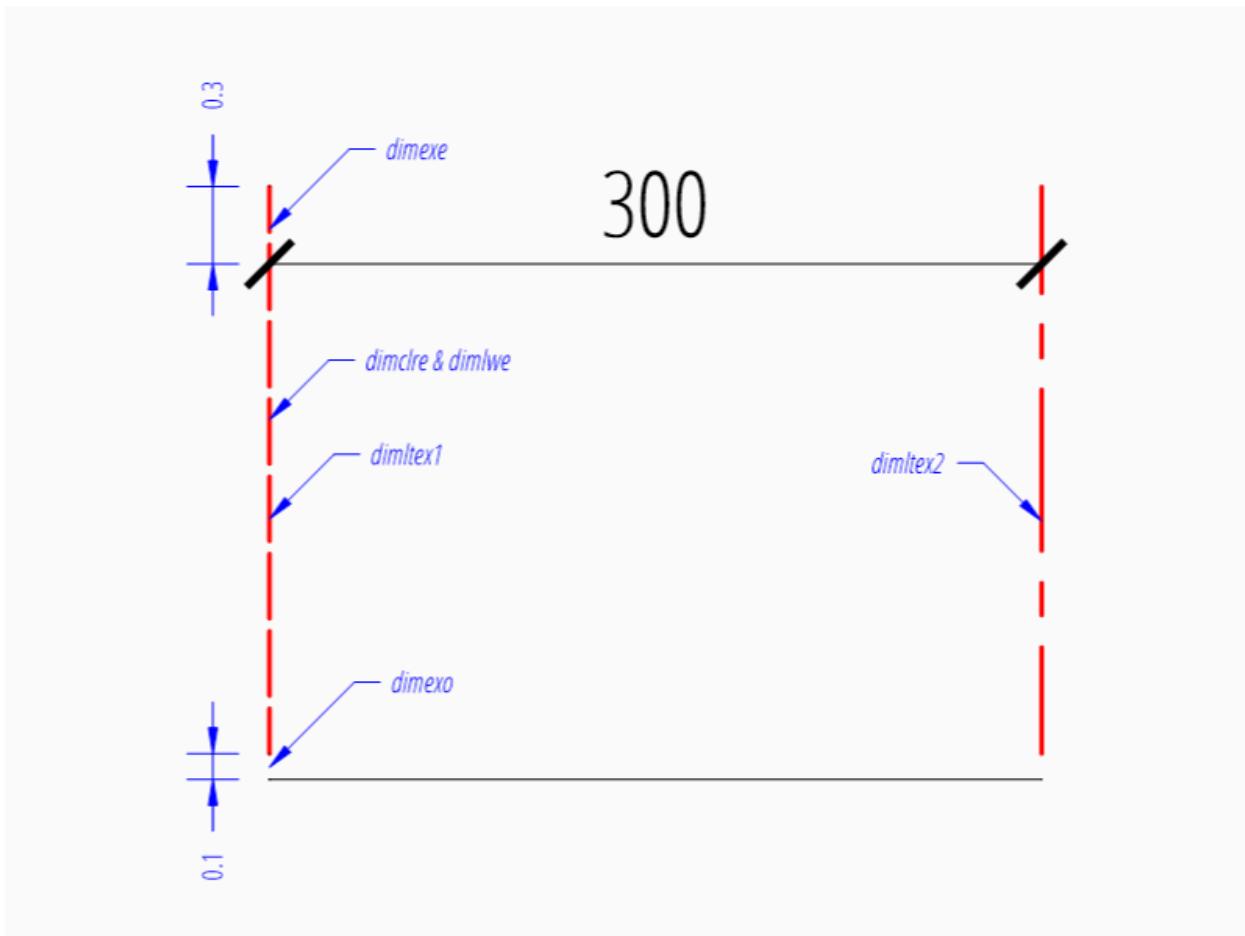
Extension Line Properties

The *extension line color* is defined by the DIMVAR `dimclre` as *AutoCAD Color Index (ACI)*. The *linetype* for the first and the second extension line is defined by `dimltx1` and `dimltx2` and requires DXF R2007. The *linewidth* is defined by `dimlwe` and required DXF R2000, see also the `linewidth` reference for valid values.

The `dimexe` is the extension of the extension line beyond the dimension line, and `dimexo` defines the offset of the extension line from the measurement point.

DIMVAR	Description
dimclre	extension line color as <i>AutoCAD Color Index (ACI)</i>
dimltex1	linetype of first extension line
dimltex2	linetype of second extension line
dimlwe	line weight of extension line
dimexe	extension beyond dimension line in drawing units
dimexo	offset of extension line from measurement point
dimfxlon	set to 1 to enable fixed length extension line
dimfxl	length of fixed length extension line in drawing units
dimse1	suppress first extension line if 1
dimse2	suppress second extension line if 1

```
msp.add_linear_dim(  
    base=(3, 2),  
    p1=(3, 0),  
    p2=(6, 0),  
    override={  
        "dimclre": 1,      # red  
        "dimltex1": "DASHED2",  
        "dimltex2": "CENTER2",  
        "dimlwe": 35,     # 0.35mm line weight  
        "dimexe": 0.3,    # length above dimension line  
        "dimexo": 0.1,    # offset from measurement point  
    }  
).render()
```

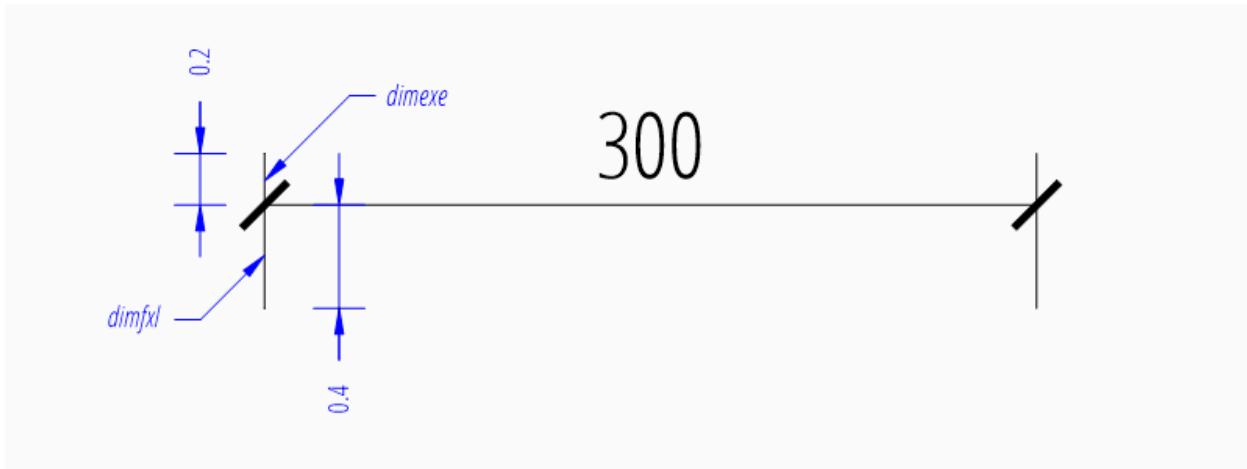


`DimStyleOverride()` methods:

```
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_extline_format(color=1, linewidth=35, extension=0.3, offset=0.1)
dim.set_extline1(linetype="DASHED2")
dim.set_extline2(linetype="CENTER2")
dim.render()
```

Fixed length extension lines are supported in DXF R2007, set `dimfxlon` to 1 and `dimfxl` defines the length of the extension line starting at the dimension line.

```
msp.add_linear_dim(
    base=(3, 2),
    p1=(3, 0),
    p2=(6, 0),
    override={
        "dimfxlon": 1,    # fixed length extension lines
        "dimexe": 0.2,   # length above dimension line
        "dimfxl": 0.4,   # length below dimension line
    }
).render()
```



`DimStyleOverride()` method:

```
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_extline_format(extension=0.2, fixed_length=0.4)
dim.render()
```

To suppress extension lines set `dimse1` to 1 to suppress the first extension line and `dimse2` to 1 to suppress the second extension line.

```
msp.add_linear_dim(
    base=(3, 2),
    p1=(3, 0),
    p2=(6, 0),
    override={
        "dimse1": 1, # suppress first extension line
        "dimse2": 1, # suppress second extension line
        "dimblk": ezdxf.ARROWS.closed_filled, # arrows just looks better
    }
).render()
```



`DimStyleOverride()` methods:

```
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_arrows(blk=ezdxf.ARROWS.closed_filled)
dim.set_extline1(disable=True)
dim.set_extline2(disable=True)
dim.render()
```

Arrows

“Arrows” mark then beginning and the end of a dimension line, and most of them do not look like arrows.

DXF distinguish between the simple tick (a slanted line) and arrows as blocks.

To use a simple tick as “arrow” set `dimtsz` to a value greater than 0, this also disables arrow blocks as side effect:

```
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_tick(size=0.25)
dim.render()
```

*Ezdx*f uses the “ARCTICK” block at double size to render the tick (AutoCAD and BricsCad just draw a simple line), so there is no advantage of using the tick instead of an arrow.

Using arrows:

```
dim = msp.add_linear_dim(base=(3, 2), p1=(3, 0), p2=(6, 0))
dim.set_arrow(blk="OPEN_30", size=0.25)
dim.render()
```

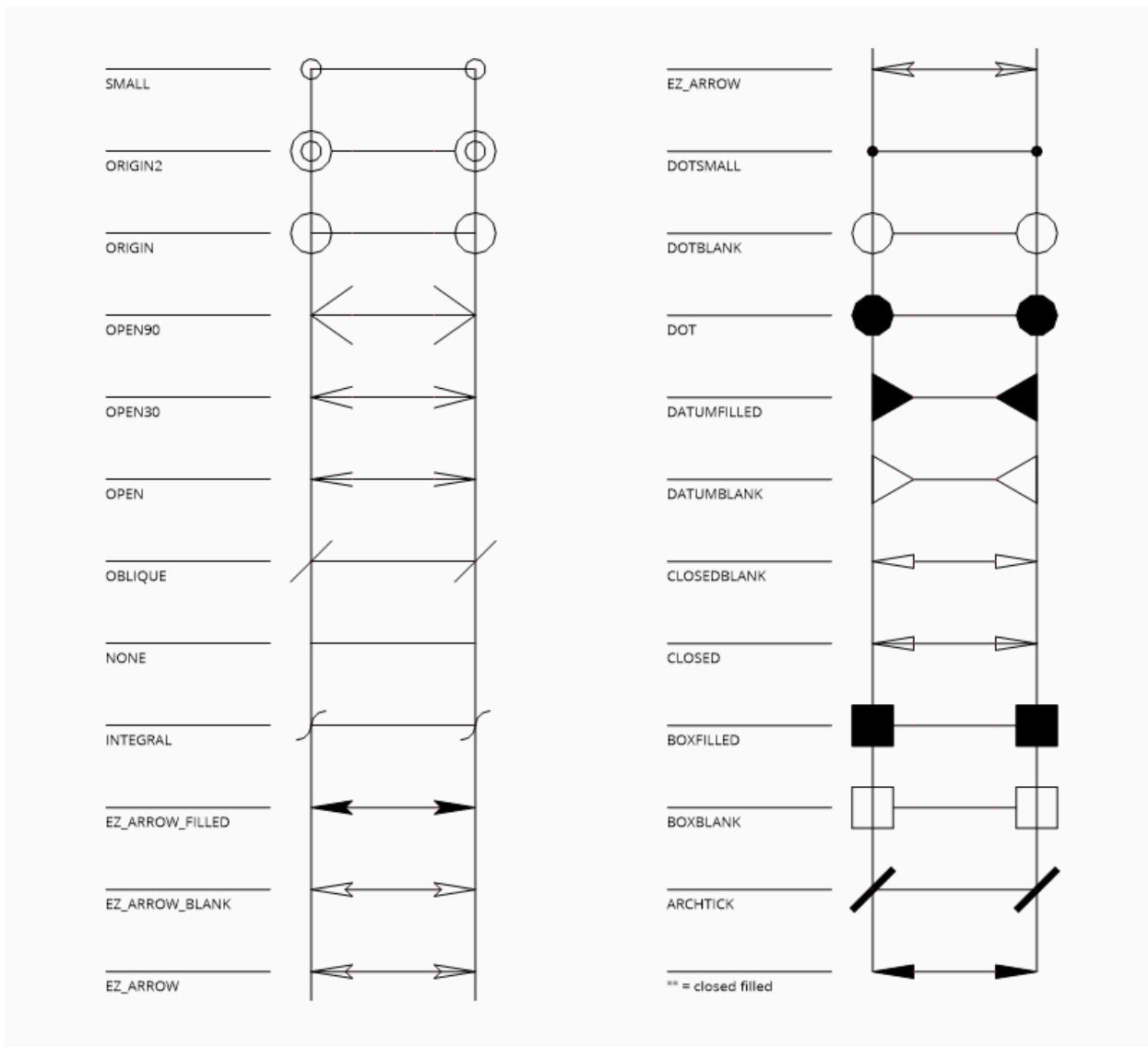
DIMVAR	Description
dimtsz	tick size in drawing units, set to 0 to use arrows
dimblk	set both arrow block names at once
dimblk1	first arrow block name
dimblk2	second arrow block name
dimasz	arrow size in drawing units

```
msp.add_linear_dim(
    base=(3, 2),
    p1=(3, 0),
    p2=(6, 0),
    override={
        "dimtsz": 0, # set tick size to 0 to enable arrow usage
        "dimasz": 0.25, # arrow size in drawing units
        "dimblk": "OPEN_30", # arrow block name
    }
).render()
```

The dimension line extension (`dimdle`) works only for a few arrow blocks and the simple tick:

- “ARCTICK”
- “OBLIQUE”
- “NONE”
- “SMALL”
- “DOTSMALL”
- “INTEGRAL”

Arrow Shapes



Arrow Names

The arrow names are stored as attributes in the `ezdxf.ARROWS` object.

closed_filled	"" (empty string)
dot	"DOT"
dot_small	"DOTSMALL"
dot_blank	"DOTBLANK"
origin_indicator	"ORIGIN"
origin_indicator_2	"ORIGIN2"
open	"OPEN"
right_angle	"OPEN90"
open_30	"OPEN30"
closed	"CLOSED"
dot_smallblank	"SMALL"
none	"NONE"
oblique	"OBIQUE"
box_filled	"BOXFILLED"
box	"BOXBLANK"
closed_blank	"CLOSEDBLANK"
datum_triangle_filled	"DATUMFILLED"
datum_triangle	"DATUMBLANK"
integral	"INTEGRAL"
architectural_tick	"ARCTICK"
ez_arrow	"EZ_ARROW"
ez_arrow_blank	"EZ_ARROW_BLANK"
ez_arrow_filled	"EZ_ARROW_FILLED"

Tolerances and Limits

The tolerances and limits features are implemented by using inline codes for the `MText` entity, therefore DXF R2000 is required. It is not possible to use both tolerances and limits at the same time.

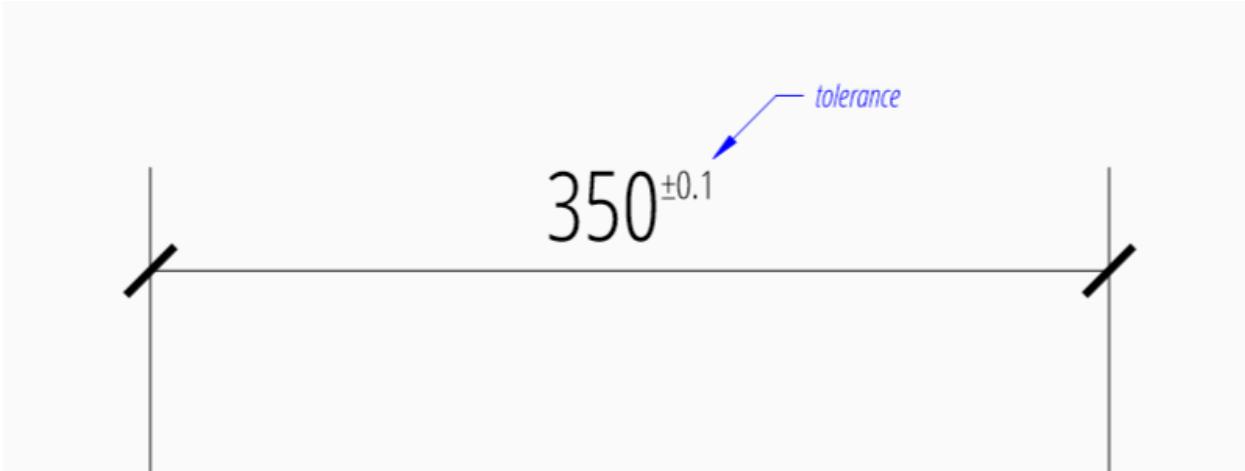
Tolerances

Geometrical tolerances are shown as additional text appended to the measurement text. It is recommend to use `set_tolerance()` method in `DimStyleOverride` or `DimStyle`.

The attribute `dimtp` defines the upper tolerance value, `dimtm` defines the lower tolerance value if present, else the lower tolerance value is the same as the upper tolerance value. Tolerance values are shown as given!

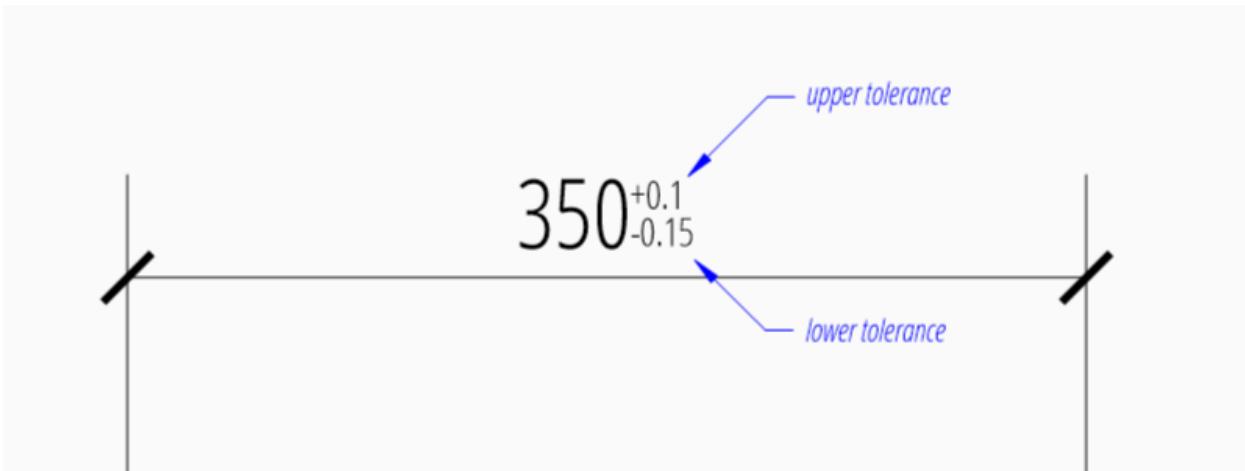
Same upper and lower tolerance value:

```
dim = msp.add_linear_dim(base=(0, 3), p1=(3, 0), p2=(6.5, 0))
dim.set_tolerance(.1, hfactor=.4, align="top", dec=2)
dim.render()
```



Different upper and lower tolerance values:

```
dim = msp.add_linear_dim(base=(0, 3), p1=(3, 0), p2=(6.5, 0))
dim.set_tolerance(upper=.1, lower=-.15, hfactor=.4, align="middle", dec=2)
dim.render()
```



The attribute `dimtfac` specifies a scale factor for the text height of limits and tolerance values relative to the dimension text height, as set by `dimtxt`. For example, if `dimtfac` is set to 1.0, the text height of fractions and tolerances is the same height as the dimension text. If `dimtxt` is set to 0.75, the text height of limits and tolerances is three-quarters the size of dimension text.

Vertical justification for tolerances is specified by `dimtolj`:

<code>dimtolj</code>	Description
0	Align with bottom line of dimension text
1	Align vertical centered to dimension text
2	Align with top line of dimension text

DIM-VAR	Description
dimtol	set to 1 to enable tolerances
dimtp	set the maximum (or upper) tolerance limit for dimension text
dimtm	set the minimum (or lower) tolerance limit for dimension text
dimtfac	specifies a scale factor for the text height of limits and tolerance values relative to the dimension text height, as set by dimtxt.
dimtzin	4 to suppress leading zeros, 8 to suppress trailing zeros or 12 to suppress both, like dimzin for dimension text, see also Text Formatting
dimtolj	set the vertical justification for tolerance values relative to the nominal dimension text.
dimec	set the number of decimal places to display in tolerance values

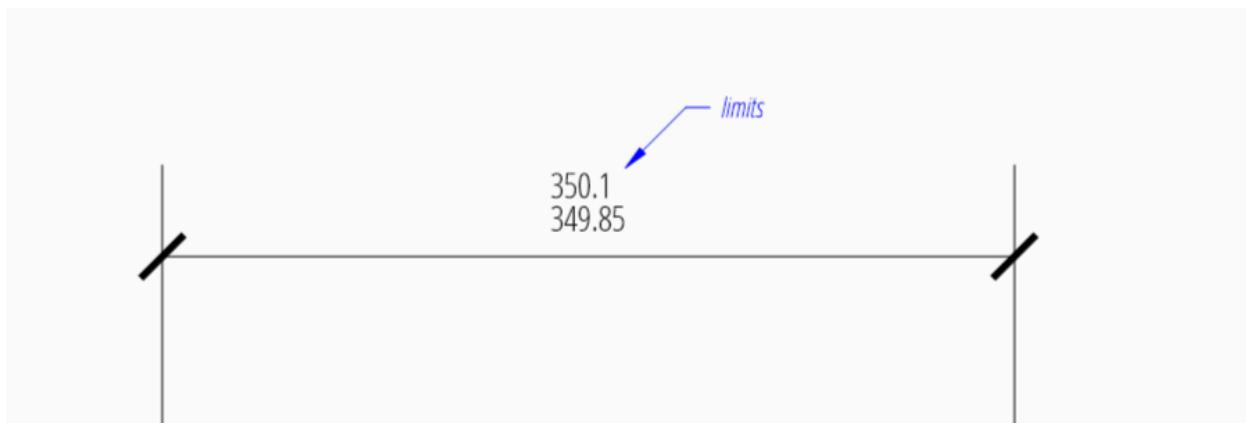
Limits

The geometrical limits are shown as upper and lower measurement limit and replaces the usual measurement text. It is recommended to use `set_limits()` method in `DimStyleOverride` or `DimStyle`.

For limits the tolerance values are drawing units scaled by measurement factor `dimlfac`, the upper limit is scaled measurement value + `dimtp` and the lower limit is scaled measurement value - `dimtm`.

The attributes `dimtfac`, `dimtzin` and `dimec` have the same meaning for limits as for tolerances.

```
dim = msp.add_linear_dim(base=(0, 3), p1=(3, 0), p2=(6.5, 0))
dim.set_limits(upper=.1, lower=.15, hfactor=.4, dec=2)
dim.render()
```



DIMVAR	Description
dimlim	set to 1 to enable limits

Alternative Units

Alternative units are not supported.

6.5.20 Tutorial for Radius Dimensions

Please read the [Tutorial for Linear Dimensions](#) before, if you haven't.

Note: *Ezdx*f does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

```
import ezdxf

# DXF R2010 drawing, official DXF version name: 'AC1024',
# setup=True setups the default dimension styles
doc = ezdxf.new("R2010", setup=True)

msp = doc.modelspace()    # add new dimension entities to the modelspace
msp.add_circle((0, 0), radius=3)  # add a CIRCLE entity, not required
# add default radius dimension, measurement text is located outside
dim = msp.add_radius_dim(
    center=(0, 0), radius=3, angle=45, dimstyle="EZ_RADIUS"
)
# necessary second step, to create the BLOCK entity with the dimension geometry.
dim.render()
doc.saveas("radius_dimension.dxf")
```

The example above creates a 45 degrees slanted radius *Dimension* entity, the default dimension style “EZ_RADIUS” is defined as 1 drawing unit = 1m, drawing scale = 1:100 and the length factor = 100, which creates a measurement text in cm, the default location for the measurement text is outside of the circle.

The *center* point defines the the center of the circle but there doesn’t have to exist a circle entity, *radius* defines the circle radius, which is also the measurement, and *angle* defines the slope of the dimension line, it is also possible to define the circle by a measurement point *mpoint* on the circle.

The return value *dim* is **not** a dimension entity, instead a *DimStyleOverride* object is returned, the dimension entity is stored as *dim.dimension*.

Placing Measurement Text

There are different predefined DIMSTYLES to achieve various text placing locations.

The basic DIMSTYLE “EZ_RADIUS” settings are:

- 1 drawing unit = 1m
- scale 1:100
- the length factor *dimlfac* = 100, which creates a measurement text in cm.
- uses a closed filled arrow, arrow size *dimasz* = 0.25

Note: Not all possibles features of DIMSTYLE are supported by the *ezdx*f rendering procedure and especially for the radial dimension there are less features implemented than for the linear dimension because of the lack of good documentation.

See also:

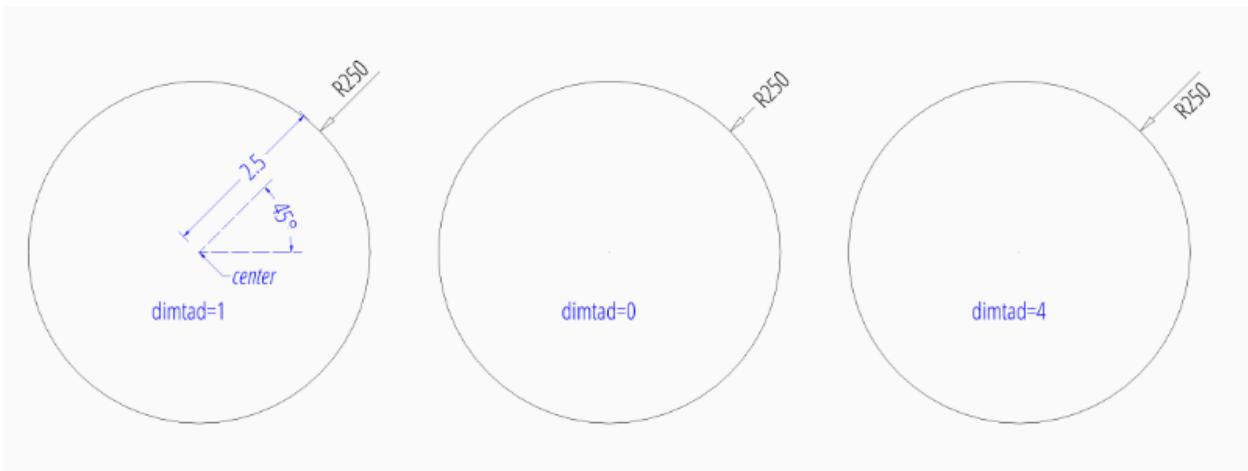
- Graphical reference of many DIMVARS and some advanced information: *DIMSTYLE Table*
- Source code file *standards.py* shows how to create your own DIMSTYLES.
- The Script *dimension_radius.py* shows examples for radius dimensions.

Default Text Locations Outside

Advanced “EZ_RADIUS” settings for placing the text outside of the circle:

tmove	1 = add a leader when dimension text is moved, this is the best setting for text outside to preserve the appearance of the DIMENSION entity, if editing afterwards in a CAD application.
dimtad	1 = place the text vertical above the dimension line

```
dim = msp.add_radius_dim(
    center=(0, 0),
    radius=2.5,
    angle=45,
    dimstyle="EZ_RADIUS"
)
dim.render()  # always required, but not shown in the following examples
```



To force text outside horizontal set `dimtoh` to 1:

```
dim = msp.add_radius_dim(
    center=(0, 0),
    radius=2.5,
    angle=45,
    dimstyle="EZ_RADIUS",
    override={"dimtoh": 1}
)
```



Default Text Locations Inside

DIMSTYLE “EZ_RADIUS_INSIDE” can be used to place the dimension text inside the circle at a default location.

The basic DIMSTYLE “EZ_RADIUS_INSIDE” settings are:

- 1 drawing unit = 1m
- scale 1:100, length_factor is 100 which creates
- the length factor dimlfac = 100, which creates a measurement text in cm.
- uses a closed filled arrow, arrow size dimasz = 0.25

Advanced “EZ_RADIUS_INSIDE” settings to place (force) the text inside of the circle:

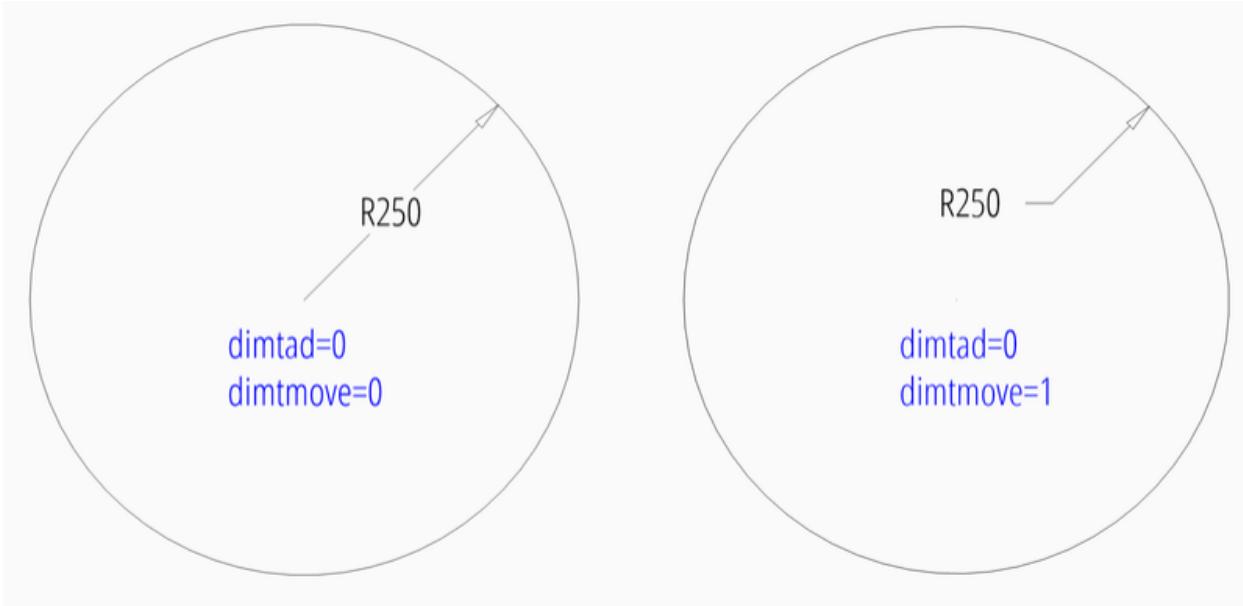
tmove	0 = moves the dimension line with dimension text, this is the best setting for text inside to preserve the appearance of the DIMENSION entity, if editing afterwards in a CAD application.
dimtix	1 = force text inside
dimmat-fit	0 = force text inside, required by BricsCAD and AutoCAD
dimtad	0 = center text vertical, BricsCAD and AutoCAD always create a vertical centered text, <i>ezdxf</i> let you choose the vertical placement (above, below, center), but editing the DIMENSION in BricsCAD or AutoCAD will reset text to center placement.

```
dim = msp.add_radius_dim(  
    center=(0, 0),  
    radius=2.5,  
    angle=45,  
    dimstyle="EZ_RADIUS_INSIDE"  
)
```



To force text inside horizontal set `dimtih` to 1:

```
dim = msp.add_radius_dim(  
    center=(0, 0),  
    radius=2.5,  
    angle=45,  
    dimstyle="EZ_RADIUS_INSIDE",  
    override={"dimtih": 1}  
)
```

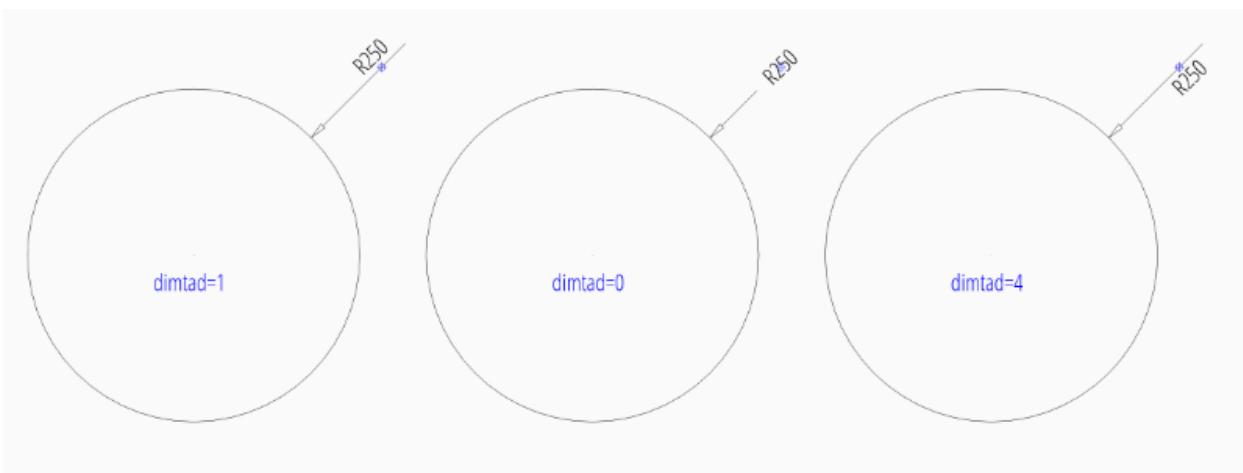


User Defined Text Locations

Beside the default location it is always possible to override the text location by a user defined location. This location also determines the angle of the dimension line and overrides the argument *angle*. For user defined locations it is not necessary to force text inside (*dimtix*=1), because the location of the text is explicit given, therefore the DIMSTYLE “EZ_RADIUS” can be used for all this examples.

User defined location outside of the circle:

```
dim = msp.add_radius_dim(  
    center=(0, 0),  
    radius=2.5,  
    location=(4, 4),  
    dimstyle="EZ_RADIUS"  
)
```



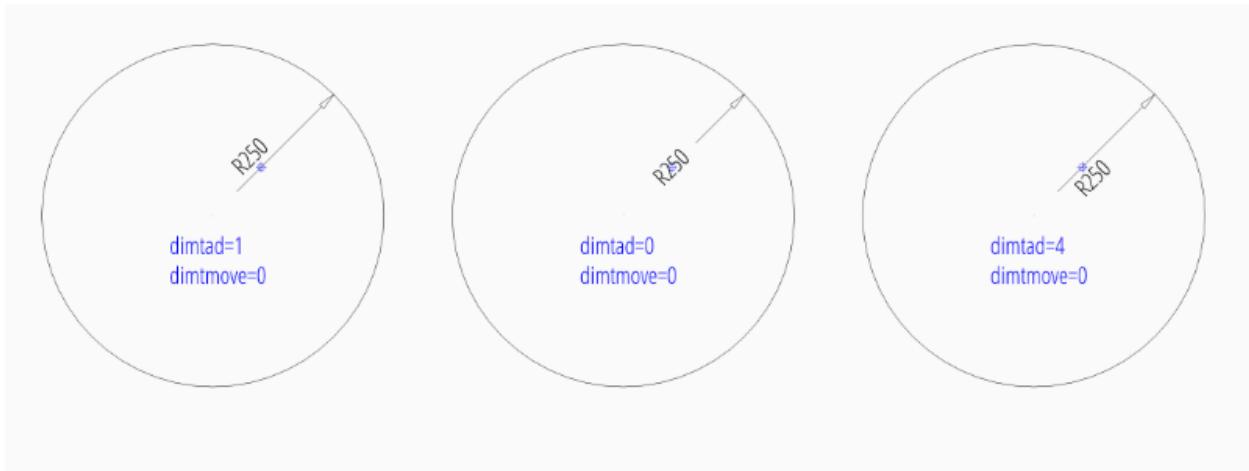
User defined location outside of the circle and forced horizontal text:

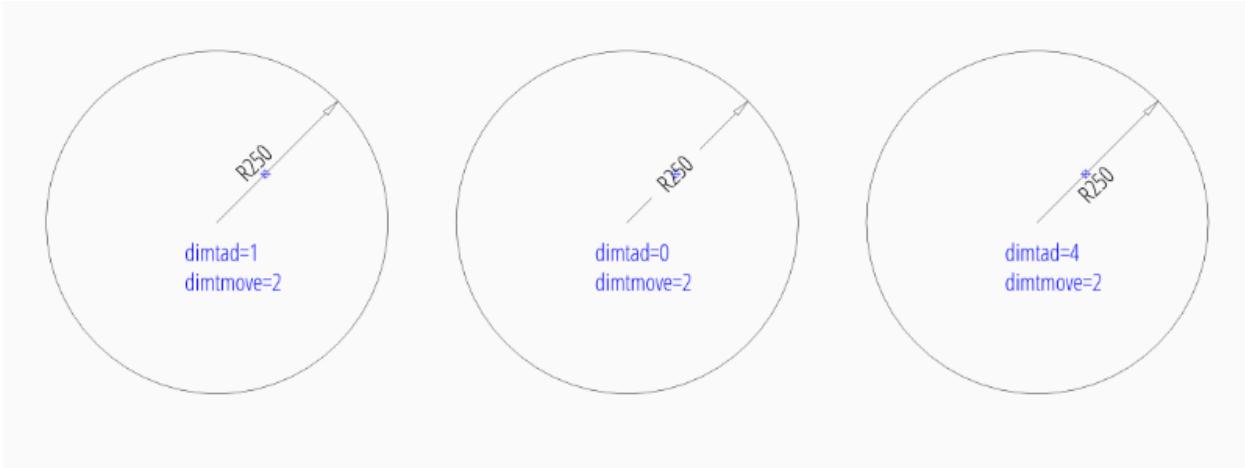
```
dim = msp.add_radius_dim(
    center=(0, 0),
    radius=2.5,
    location=(4, 4),
    dimstyle="EZ_RADIUS",
    override={"dimtoh": 1}
)
```



User defined location inside of the circle:

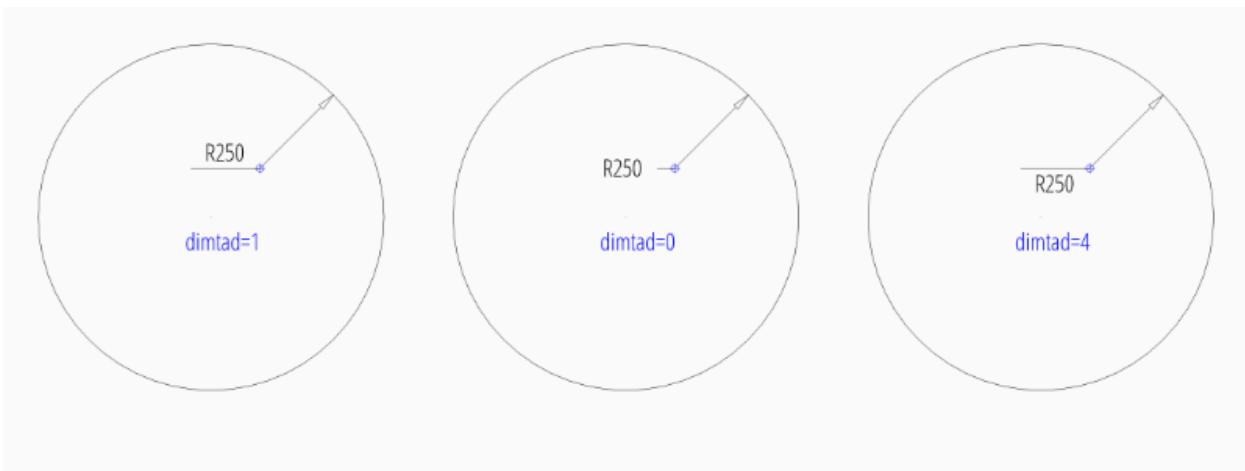
```
dim = msp.add_radius_dim(
    center=(0, 0),
    radius=2.5,
    location=(1, 1),
    dimstyle="EZ_RADIUS"
)
```





User defined location inside of the circle and forced horizontal text:

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



Center Mark/Lines

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

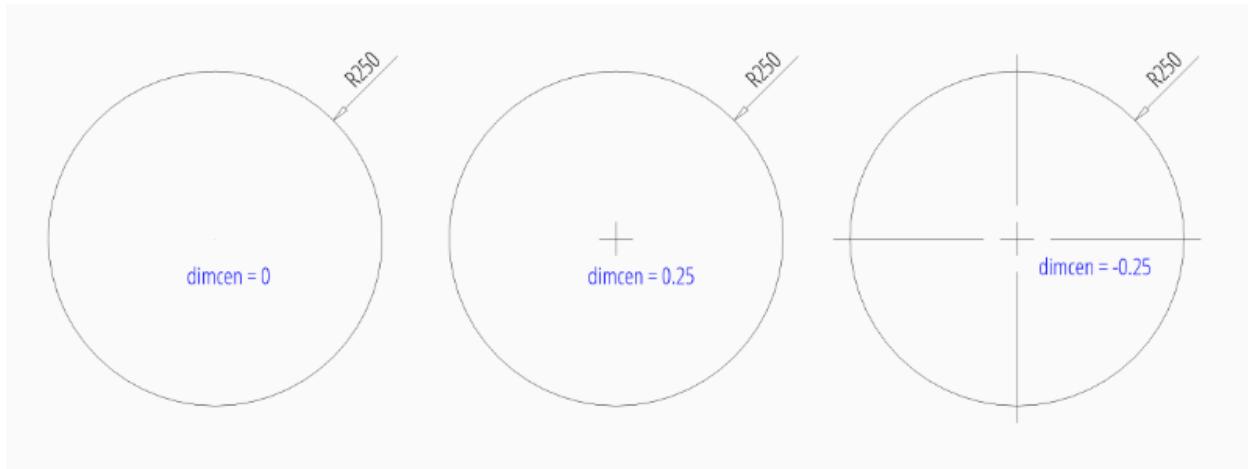
0	Center mark is off
>0	Create center mark of given size
<0	Create center lines

```
dim = msp.add_radius_dim(
    center=(0, 0),
```

(continues on next page)

(continued from previous page)

```
radius=2.5,
angle=45,
dimstyle="EZ_RADIUS",
override={"dimcen": 0.25},
)
```



Overriding Measurement Text

See Linear Dimension Tutorial: [Overriding Measurement Text](#)

Measurement Text Formatting and Styling

See Linear Dimension Tutorial: [Measurement Text Formatting and Styling](#)

6.5.21 Tutorial for Diameter Dimensions

Please read the [Tutorial for Radius Dimensions](#) before, if you haven't.

Note: Ezdxf does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

This is a repetition of the radius tutorial, just with diameter dimensions.

```
import ezdxf

# setup=True setups the default dimension styles
doc = ezdxf.new("R2010", setup=True)

msp = doc.modelspace() # add new dimension entities to the modelspace
msp.add_circle((0, 0), radius=3) # add a CIRCLE entity, not required
# add default diameter dimension, measurement text is located outside
dim = msp.add_diameter_dim(
    center=(0, 0),
    radius=3,
    angle=45,
    dimstyle="EZ_RADIUS"
)
```

(continues on next page)

(continued from previous page)

```
)
dim.render()
doc.saveas("diameter_dimension.dxf")
```

The example above creates a 45 degrees slanted diameter *Dimension* entity, the default dimension style “EZ_RADIUS” (same as for radius dimensions) is defined as 1 drawing unit = 1m, drawing scale = 1:100 and the length factor = 100, which creates a measurement text in cm, the default location for the measurement text is outside of the circle.

The *center* point defines the the center of the circle but there doesn’t have to exist a circle entity, *radius* defines the circle radius and *angle* defines the slope of the dimension line, it is also possible to define the circle by a measurement point *mpoint* on the circle.

The return value *dim* is **not** a dimension entity, instead a *DimStyleOverride* object is returned, the dimension entity is stored as *dim.dimension*.

Placing Measurement Text

There are different predefined DIMSTYLES to achieve various text placing locations.

The basic DIMSTYLE “EZ_RADIUS” settings are:

- 1 drawing unit = 1m
- scale 1:100
- the length factor dimlfac = 100, which creates a measurement text in cm.
- uses a closed filled arrow, arrow size dimasz = 0.25

Note: Not all possible features of DIMSTYLE are supported by the *ezdxf* rendering procedure and especially for the diameter dimension there are less features implemented than for the linear dimension because of the lack of good documentation.

See also:

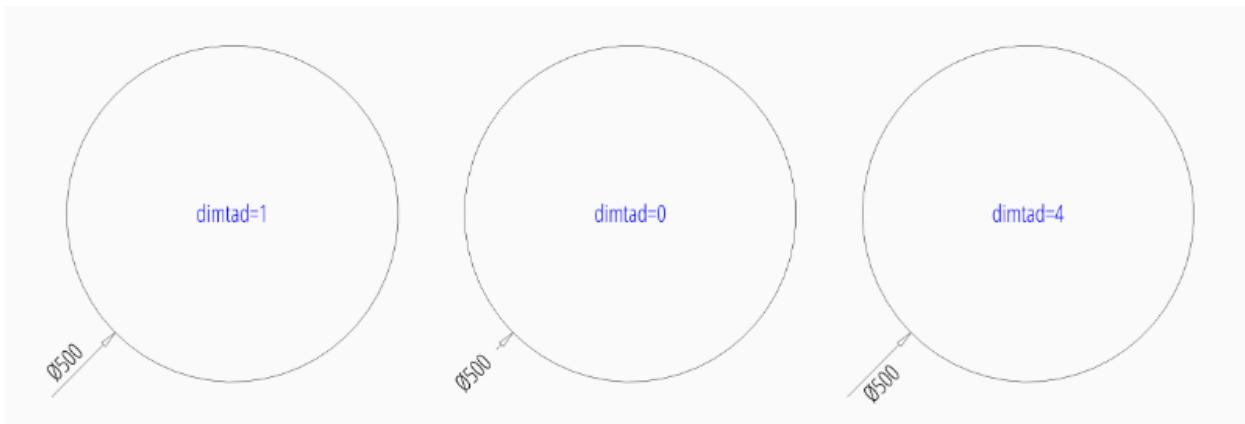
- Graphical reference of many DIMVARS and some advanced information: *DIMSTYLE Table*
- Source code file `standards.py` shows how to create your own DIMSTYLES.
- The Script `dimension_diameter.py` shows examples for radius dimensions.

Default Text Locations Outside

“EZ_RADIUS” default settings for to place text outside:

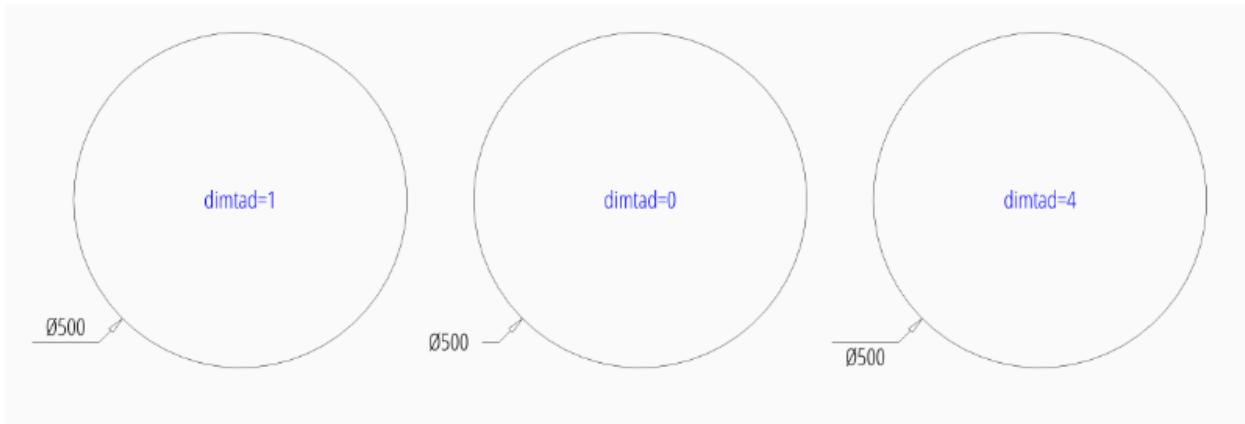
tmove	1 = add a leader when dimension text is moved, this is the best setting for text outside to preserve the appearance of the DIMENSION entity, if editing afterwards in a CAD application.
dim-tad	1 = place the text vertical above the dimension line

```
dim = msp.add_diameter_dim(
    center=(0, 0),
    radius=2.5,
    angle=45,
    dimstyle="EZ_RADIUS"
)
dim.render()  # always required, but not shown in the following examples
```



To force text outside horizontal set `dimtoh` to 1:

```
dim = msp.add_diameter_dim(
    center=(0, 0),
    radius=2.5,
    angle=45,
    dimstyle="EZ_RADIUS",
    override={"dimtoh": 1}
)
```



Default Text Locations Inside

DIMSTYLE “EZ_RADIUS_INSIDE” can be used to place the dimension text inside the circle at a default location.

The basic DIMSTYLE settings are:

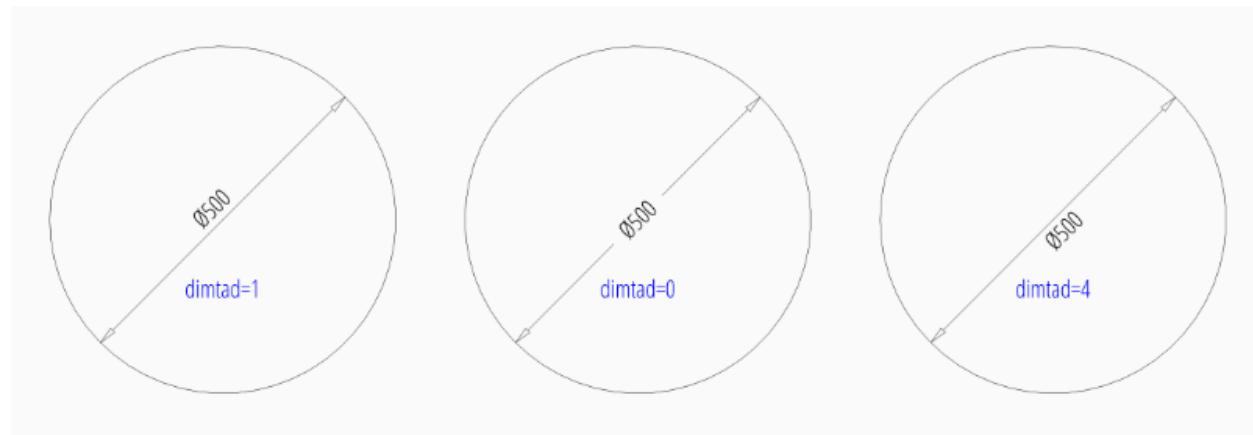
- 1 drawing unit = 1m
- scale 1:100, length_factor is 100 which creates

- the length factor `dimlfac = 100`, which creates a measurement text in cm.
- uses a closed filled arrow, arrow size `dimasz = 0.25`

Advanced “EZ_RADIUS_INSIDE” settings to place (force) the text inside of the circle:

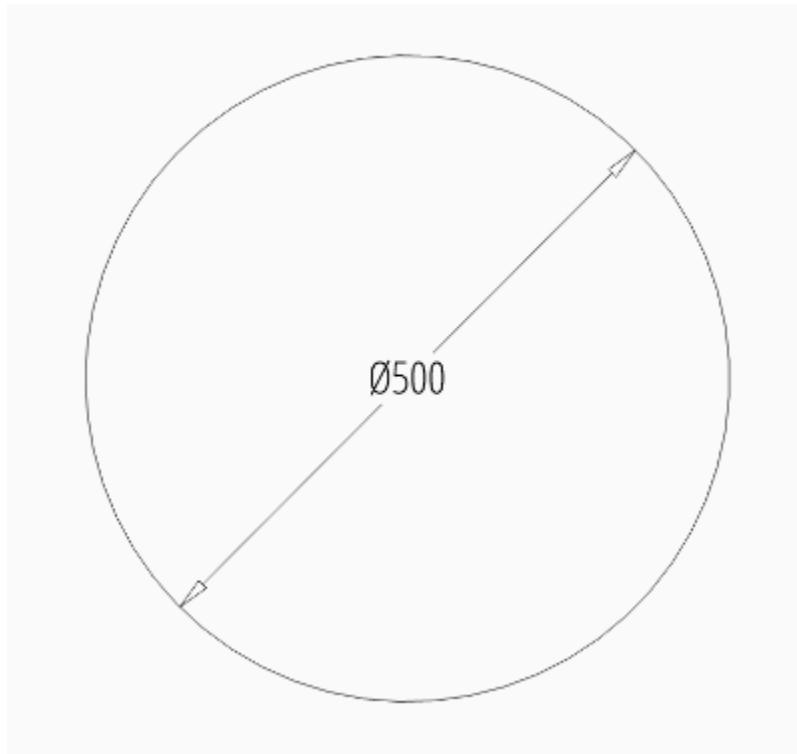
<code>tmove</code>	0 = moves the dimension line with dimension text, this is the best setting for text inside to preserve the appearance of the DIMENSION entity, if editing afterwards in a CAD application.
<code>dmtix</code>	1 = force text inside
<code>dimmatfit</code>	0 = force text inside, required by BricsCAD and AutoCAD
<code>dimtad</code>	0 = center text vertical, BricsCAD and AutoCAD always create a vertical centered text, <i>ezdxf</i> let you choose the vertical placement (above, below, center), but editing the DIMENSION in BricsCAD or AutoCAD will reset text to center placement.

```
dim = msp.add_diameter_dim(
    center=(0, 0),
    radius=2.5,
    angle=45,
    dimstyle="EZ_RADIUS_INSIDE"
)
```



To force text inside horizontal set `dmtih` to 1:

```
dim = msp.add_diameter_dim(
    center=(0, 0),
    radius=2.5,
    angle=45,
    dimstyle="EZ_RADIUS_INSIDE",
    override={"dmtih": 1}
)
```

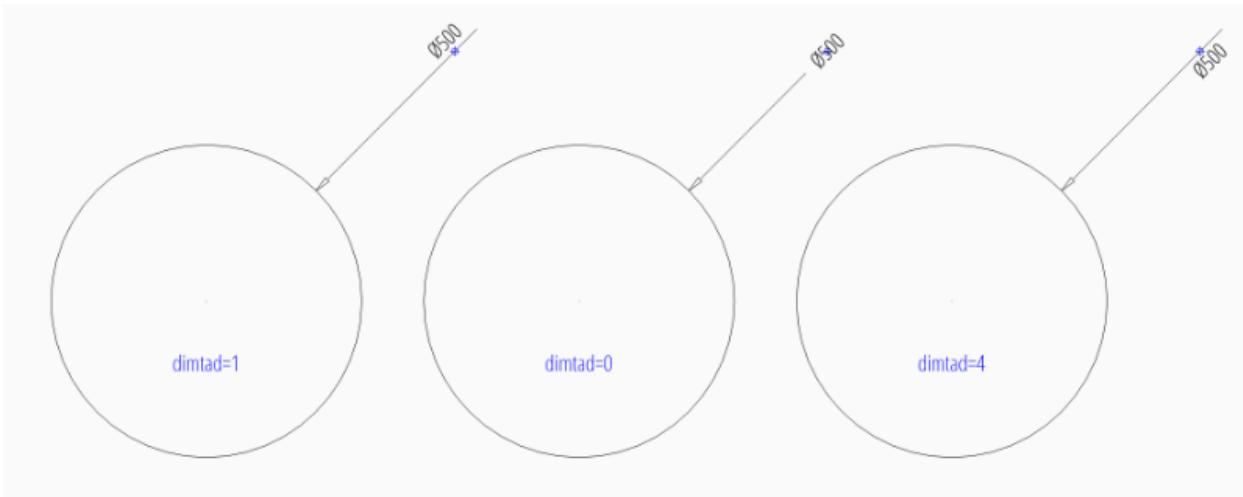


User Defined Text Locations

Beside the default location it is always possible to override the text location by a user defined location. This location also determines the angle of the dimension line and overrides the argument *angle*. For user defined locations it is not necessary to force text inside (`dimtext=1`), because the location of the text is explicit given, therefore the DIMSTYLE “EZ_RADIUS” can be used for all this examples.

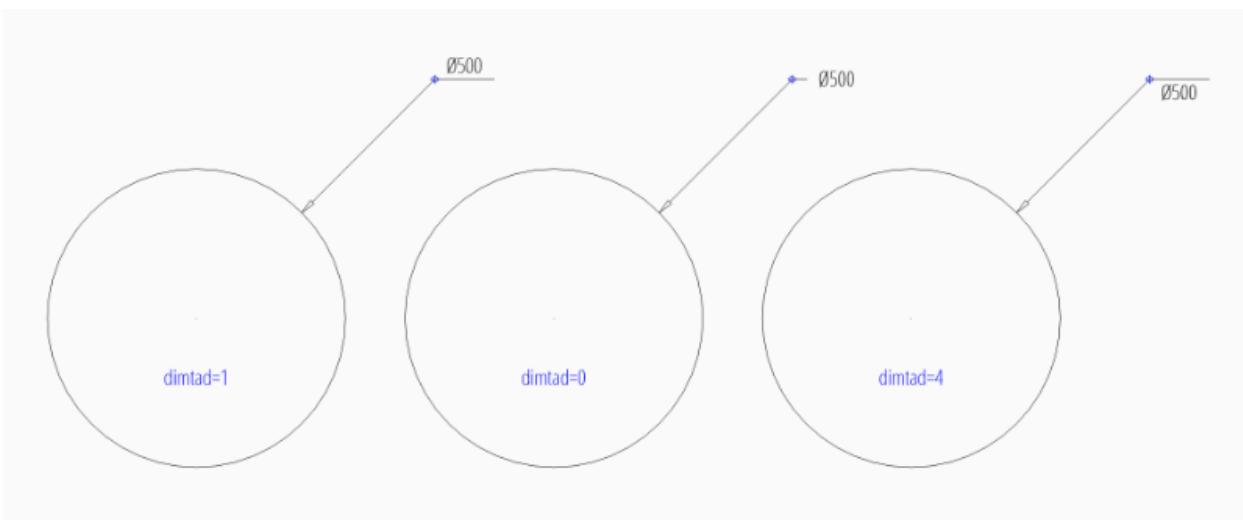
User defined location outside of the circle:

```
dim = msp.add_diameter_dim(  
    center=(0, 0),  
    radius=2.5,  
    location=(4, 4),  
    dimstyle="EZ_RADIUS"  
)
```



User defined location outside of the circle and forced horizontal text:

```
dim = msp.add_diameter_dim(  
    center=(0, 0),  
    radius=2.5,  
    location=(4, 4),  
    dimstyle="EZ_RADIUS",  
    override={"dimtoh": 1}  
)
```



User defined location inside of the circle:

```
dim = msp.add_diameter_dim(  
    center=(0, 0),  
    radius=2.5,  
    location=(1, 1),  
    dimstyle="EZ_RADIUS"  
)
```



User defined location inside of the circle and forced horizontal text:

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



Center Mark/Lines

See Radius Dimension Tutorial: [Center Mark/Lines](#)

Overriding Measurement Text

See Linear Dimension Tutorial: [Overriding Measurement Text](#)

Measurement Text Formatting and Styling

See Linear Dimension Tutorial: [Measurement Text Formatting and Styling](#)

6.5.22 Tutorial for Angular Dimensions

Please read the [Tutorial for Linear Dimensions](#) before, if you haven't.

Note: *Ezdx*f does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Dimension Style “EZ_CURVED”

All factory methods to create angular dimensions uses the dimension style “EZ_CURVED” for curved dimension lines which is defined as:

- angle unit is decimal degrees, dimaunit = 0
- measurement text height = 0.25 (drawing scale = 1:100)
- measurement text location is above the dimension line
- closed filled arrow and arrow size dimasz = 0.25
- dimazin = 2, suppresses trailing zeros (e.g. 12.5000 becomes 12.5)

This DIMENSION style only exist if the argument *setup* is True for creating a new DXF document by *ezdxf.new()*. Every dimension style which does not exist will be replaced by the dimension style “Standard” at DXF export by *save()* or *saveas()* (e.g. dimension style setup was not initiated).

Add all *ezdxf* specific resources (line types, text- and dimension styles) to an existing DXF document:

```
import ezdxf
from ezdxf.tools.standards import setup_drawing

doc = ezdxf.readfile("your.dxf")
setup_drawing(doc, topics="all")
```

Factory Methods to Create Angular Dimensions

Defined by Center, Radius and Angles

The first example shows an angular dimension defined by the center point, radius, start- and end angles:

```
import ezdxf

# Create a DXF R2010 document:
# Use argument setup=True to setup the default dimension styles.
doc = ezdxf.new("R2010", setup=True)

# Add new entities to the modelspace:
msp = doc.modelspace()

# Add an angular DIMENSION defined by the center point, start- and end angles,
# the measurement text is placed at the default location above the dimension
# line:
dim = msp.add_angular_dim_cra(
    center=(5, 5), # center point of the angle
    radius=7, # distance from center point to the start of the extension lines
    start_angle=60, # start angle in degrees
```

(continues on next page)

(continued from previous page)

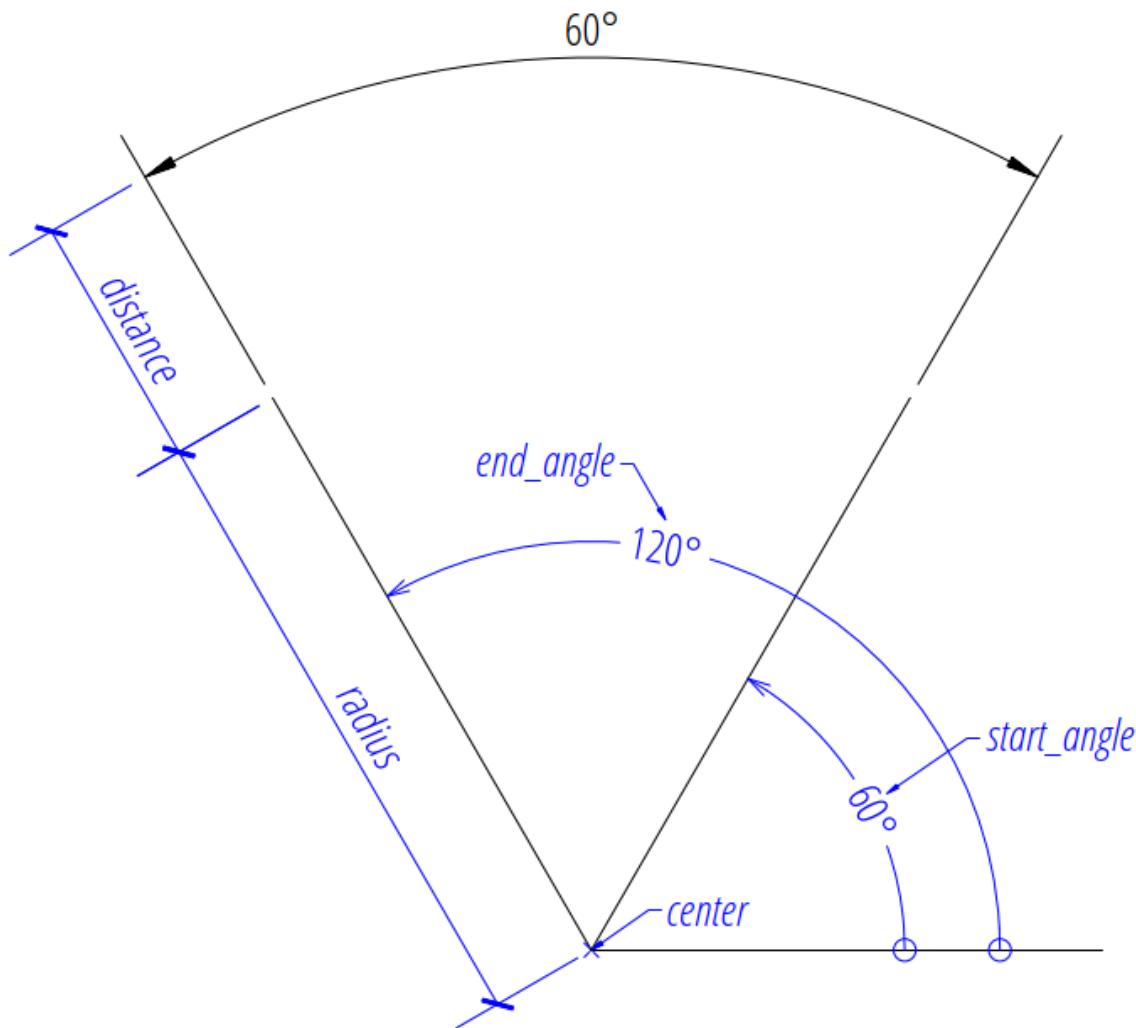
```

end_angle=120, # end angle in degrees
distance=3, # distance from start of the extension lines to the dimension line
dimstyle="EZ_CURVED", # default angular dimension style
)

# Necessary second step to create the BLOCK entity with the dimension geometry.
# Additional processing of the DIMENSION entity could happen between adding
# the entity and the rendering call.
dim.render()
doc.saveas("angular_dimension_cra.dxf")

```

The return value `dim` is **not** a dimension entity, instead a `DimStyleOverride` object is returned, the dimension entity is stored as `dim.dimension`.



Angle by 2 Lines

The next example shows an angular dimension for an angle defined by two lines:

```
import ezdxf

doc = ezdxf.new(setup=True)
msp = doc.modelspace()

# Setup the geometric parameters for the DIMENSION entity:
base = (5.8833, -6.3408) # location of the dimension line
p1 = (2.0101, -7.5156) # start point of 1st leg
p2 = (2.7865, -10.4133) # end point of 1st leg
p3 = (6.7054, -7.5156) # start point of 2nd leg
p4 = (5.9289, -10.4133) # end point of 2nd leg

# Draw the lines for visualization, not required to create the
# DIMENSION entity:
msp.add_line(p1, p2)
msp.add_line(p3, p4)

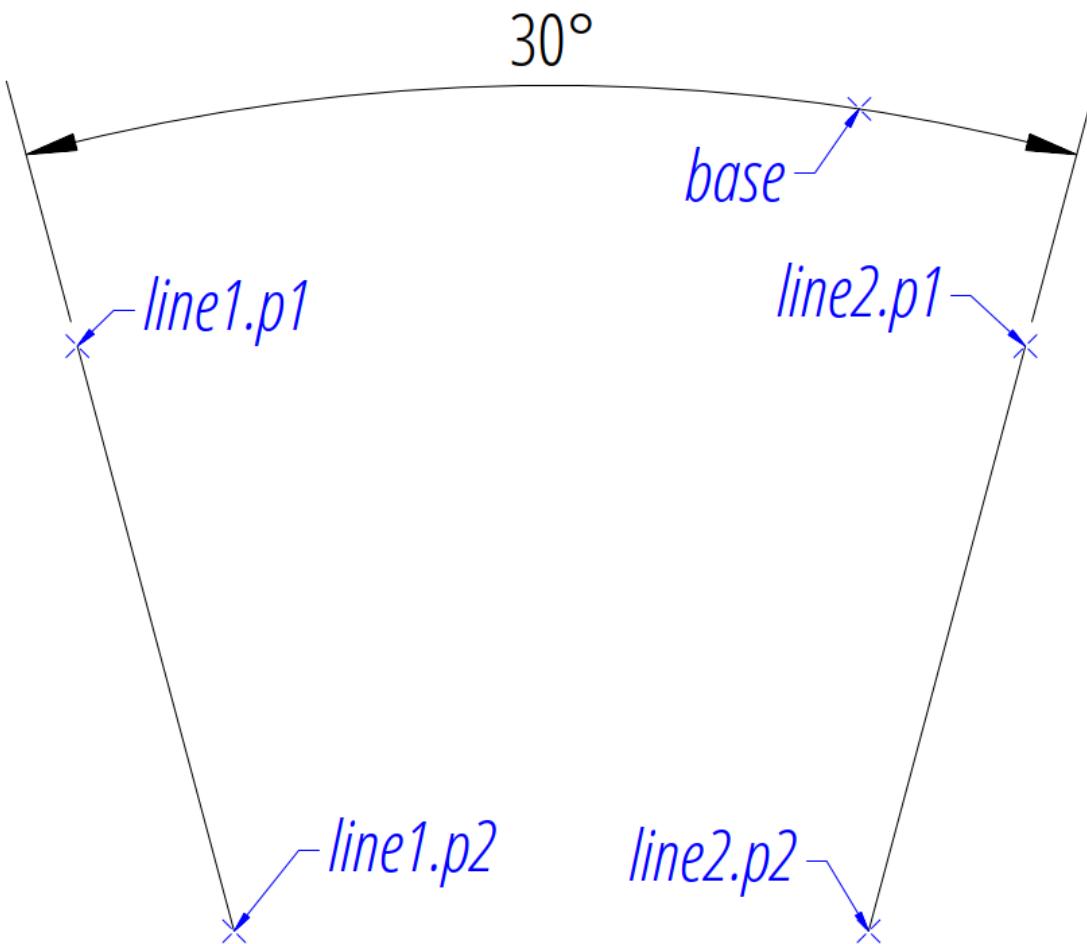
# Add an angular DIMENSION defined by two lines, the measurement text is
# placed at the default location above the dimension line:
dim = msp.add_angular_dim_2l(
    base=base, # defines the location of the dimension line
    line1=(p1, p2), # start leg of the angle
    line2=(p3, p4), # end leg of the angle
    dimstyle="EZ_CURVED", # default angular dimension style
)

# Necessary second step to create the dimension line geometry:
dim.render()
doc.saveas("angular_dimension_2l.dxf")
```

The example above creates an angular *Dimension* entity to measures the angle between two lines (*line1* and *line2*).

The *base* point defines the location of the dimension line (arc), any point on the dimension line is valid. The points *p1* and *p2* define the first leg of the angle, *p1* also defines the start point of the first extension line. The points *p3* and *p4* define the second leg of the angle and point *p3* also defines the start point of the second extension line.

The measurement of the DIMENSION entity is the angle enclosed by the first and the second leg and where the dimension line passes the *base* point.



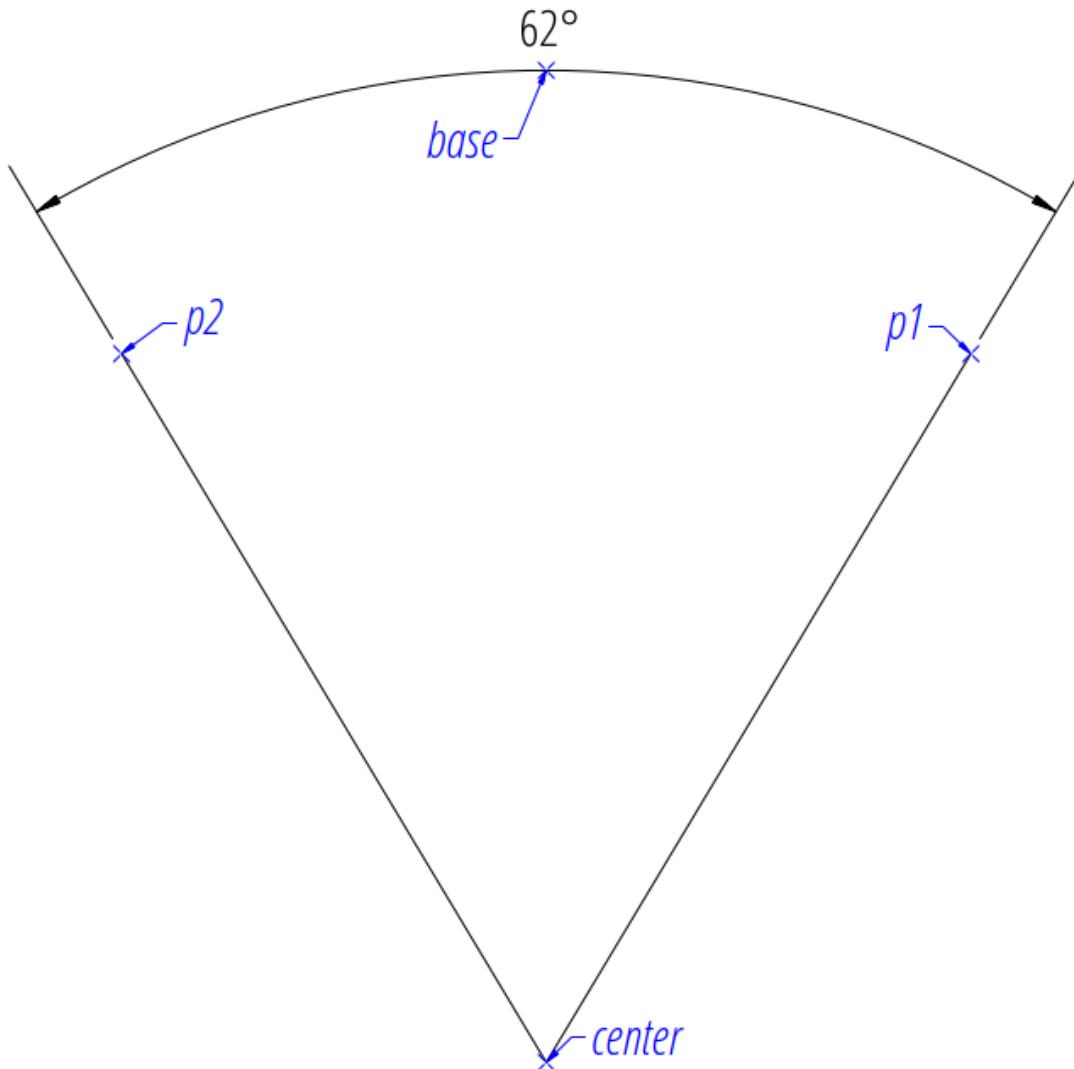
Angle by 3 Points

The next example shows an angular dimension defined by three points, a center point and the two end points of the angle legs:

```
import ezdxf

doc = ezdxf.new(setup=True)
msp = doc.modelspace()

msp.add_angular_dim_3p(
    base=(0, 7), # location of the dimension line
    center=(0, 0), # center point
    p1=(-3, 5), # end point of 1st leg = start angle
    p2=(3, 5), # end point of 2nd leg = end angle
).render()
```



Angle from ConstructionArc

The `ezdxf.math.ConstructionArc` provides various class methods for creating arcs and the construction tool can be created from an ARC entity.

Add an angular dimension to an ARC entity:

```
import ezdxf

doc = ezdxf.new(setup=True)
msp = doc.modelspace()

arc = msp.add_arc(
    center=(0, 0),
    radius=5,
    start_angle = 60,
```

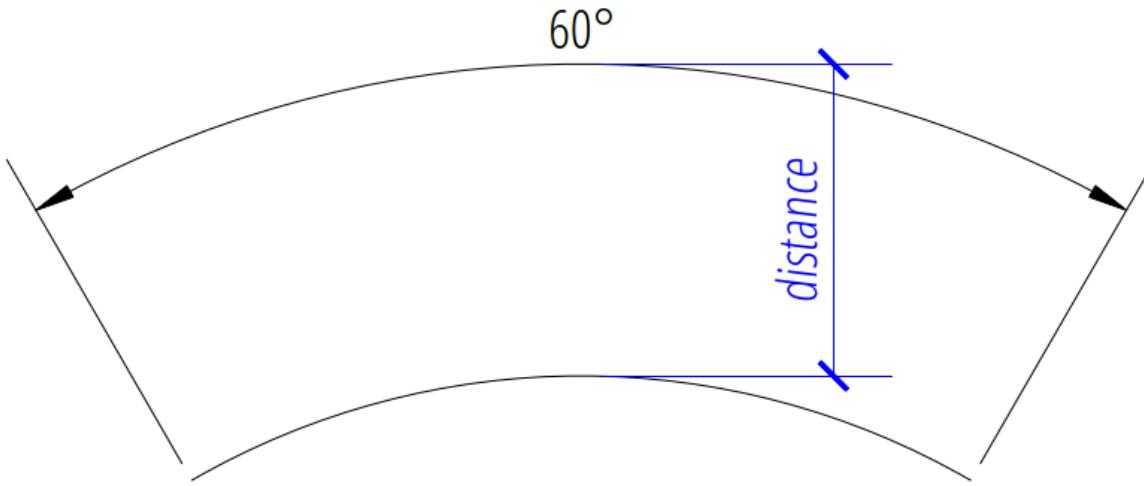
(continues on next page)

(continued from previous page)

```

        end_angle = 120,
    )
msp.add_angular_dim_arc(
    arc.construction_tool(),
    distance=2,
).render()

```



Placing Measurement Text

The default location of the measurement text depends on various *DimStyle* parameters and is applied if no user defined text location is defined.

Note: Not all possible features of DIMSTYLE are supported by the *ezdxf* rendering procedure and especially for the angular dimension there are less features implemented than for the linear dimension because of the lack of good documentation.

See also:

- Graphical reference of many DIMVARS and some advanced information: [DIMSTYLE Table](#)
- Source code file `standards.py` shows how to create your own DIMSTYLES.
- The Script `dimension_angular.py` shows examples for angular dimensions.

Default Text Locations

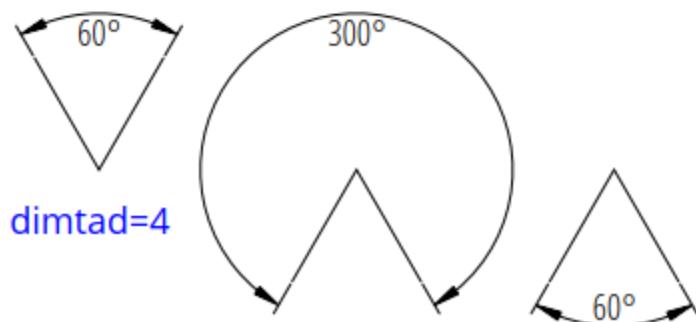
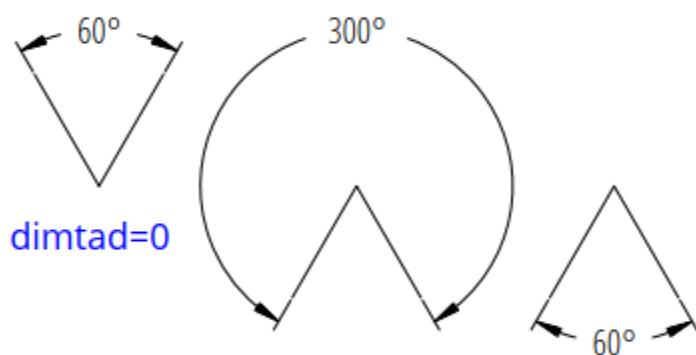
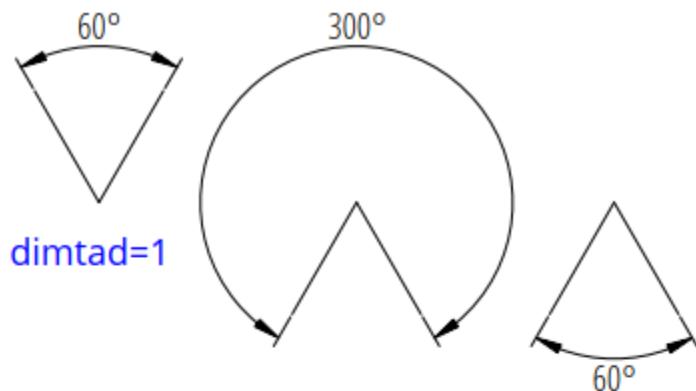
The DIMSTYLE “EZ_CURVED” places the measurement text in the center of the angle above the dimension line. The first examples above show the measurement text at the default text location.

The text direction angle is always perpendicular to the line from the text center to the center point of the angle unless this angle is manually overridden.

The “vertical” location of the measurement text relative to the dimension line is defined by `dimtad`:

0	Center, it is possible to adjust the vertical location by dimtvp
1	Above
2	Outside, handled like <i>Above</i> by <i>ezdxf</i>
3	JIS, handled like <i>Above</i> by <i>ezdxf</i>
4	Below

```
msp.add_angular_dim_cra(  
    center=(3, 3),  
    radius=3,  
    distance=1,  
    start_angle=60,  
    end_angle=120,  
    override={  
        "dimtad": 1, # 0=center; 1=above; 4=below;  
    },  
) .render()
```



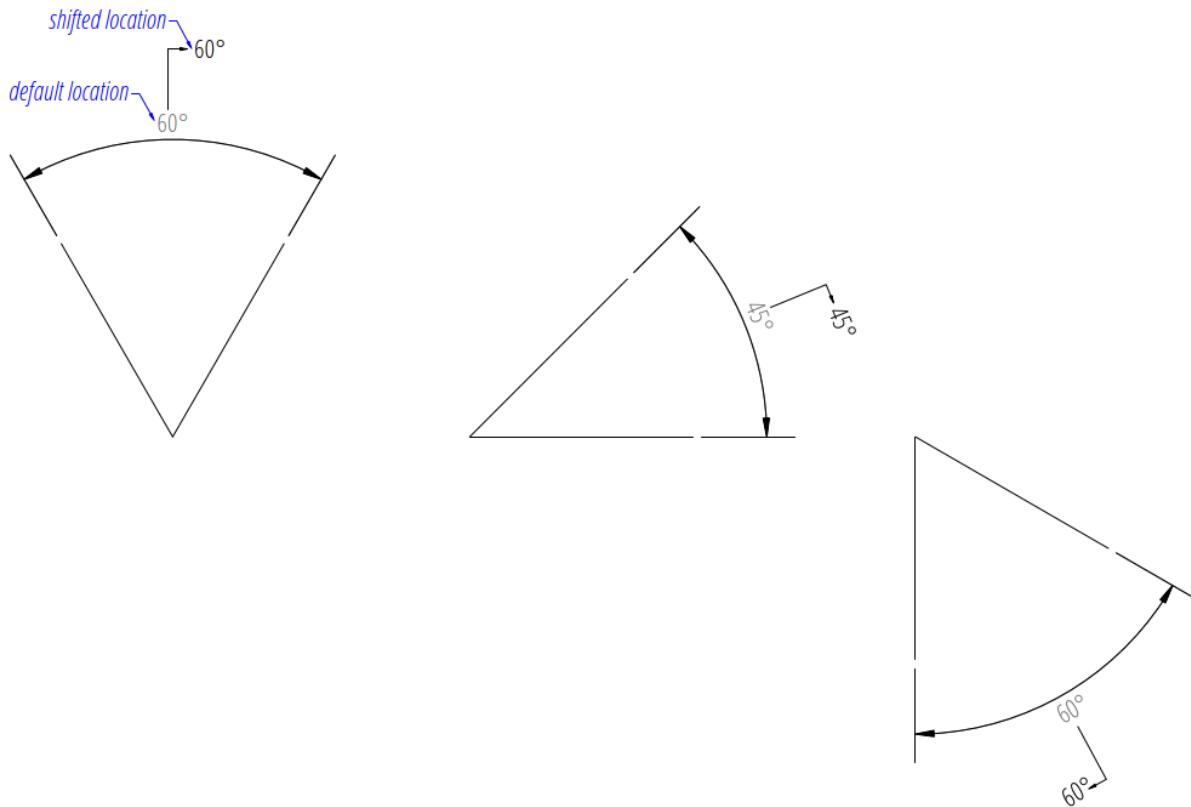
Arrows and measurement text are placed “outside” automatically if the available space between the extension lines isn’t sufficient. This overrides the `dimtad` value by 1 (“above”). *Ezdx*f follows its own rules, ignores the `dimatfit` attribute and works similar to `dimatfit = 1`, move arrows first, then text:



Shift Text From Default Location

The method `shift_text()` shifts the measurement text away from the default location. The shifting direction is aligned to the text rotation of the default measurement text.

```
dim = msp.add_angular_dim_cra(
    center=(3, 3),
    radius=3,
    distance=1,
    start_angle=60,
    end_angle=120,
)
# shift text from default text location:
dim.shift_text(0.5, 1.0)
dim.render()
```



This is just a rendering effect, editing the dimension line in a CAD application resets the text to the default location.

User Defined Text Locations

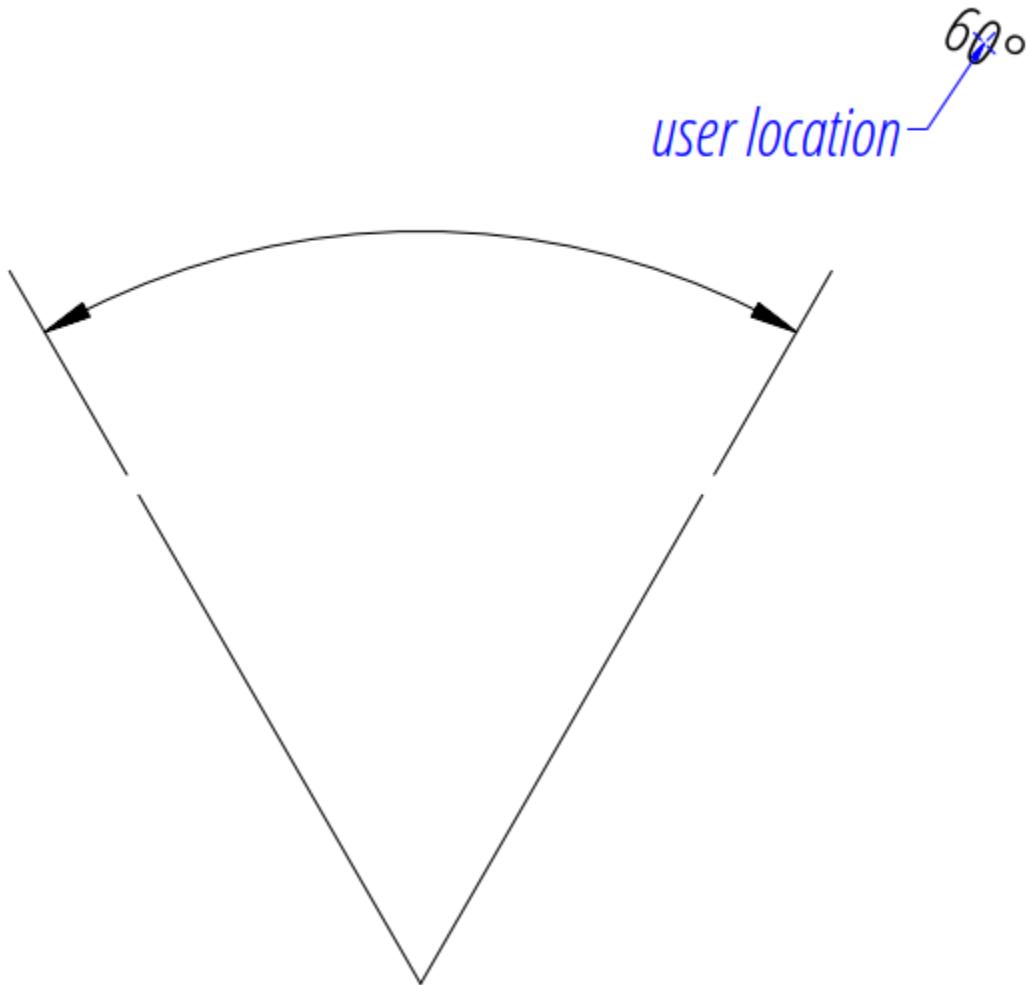
Beside the default location it is always possible to override the text location by a user defined location.

The coordinates of user locations are located in the rendering UCS and the default rendering UCS is the [WCS](#).

Absolute User Location

Absolute placing of the measurement text means relative to the origin of the render UCS. The user location is stored in the DIMENSION entity, which means editing the dimension line in a CAD application does not alter the text location. This location also determines the rotation of the measurement text.

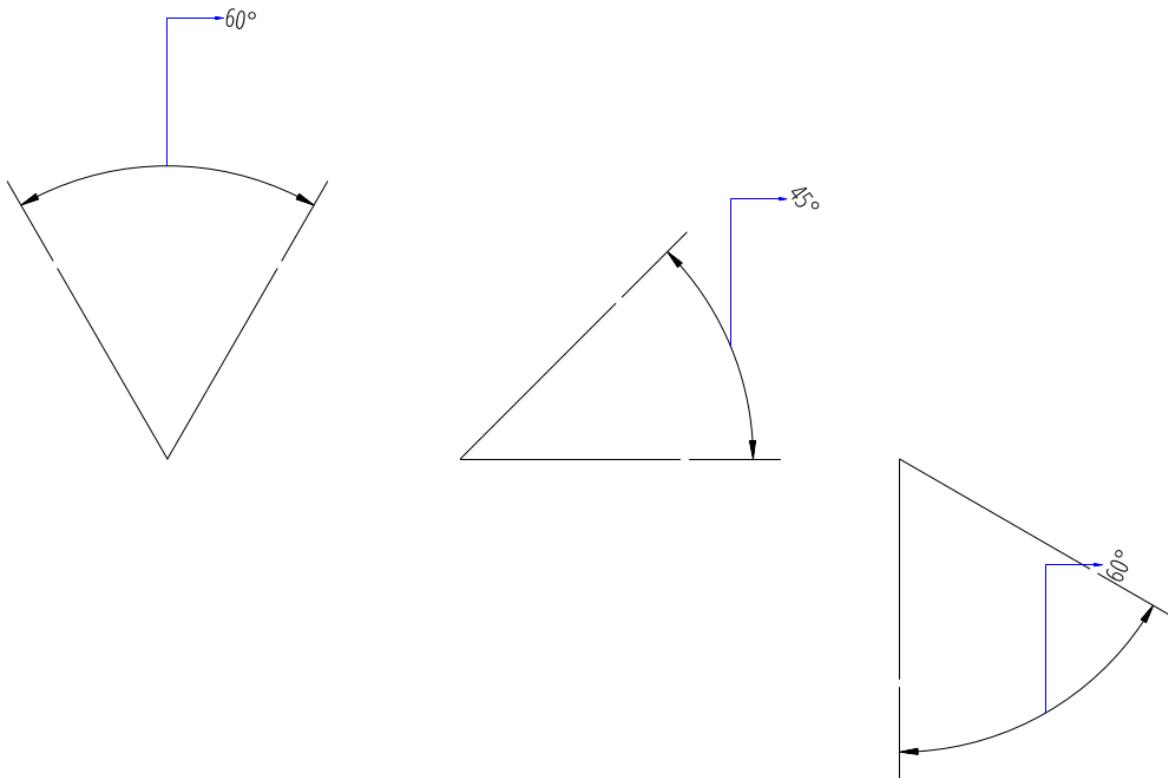
```
dim = msp.add_angular_dim_cra(  
    center=(3, 3),  
    radius=3,  
    distance=1,  
    start_angle=60,  
    end_angle=120,  
    location=(5, 8),  # user defined measurement text location  
)  
dim.render()
```



Relative User Location

Relative placing of the measurement text means relative to the middle of the dimension line. This is only possible by calling the `set_location()` method, and the argument `relative` has to be `True`. The user location is stored in the `DIMENSION` entity, which means editing the dimension line in a CAD application does not alter the text location. This location also determines the rotation of the measurement text.

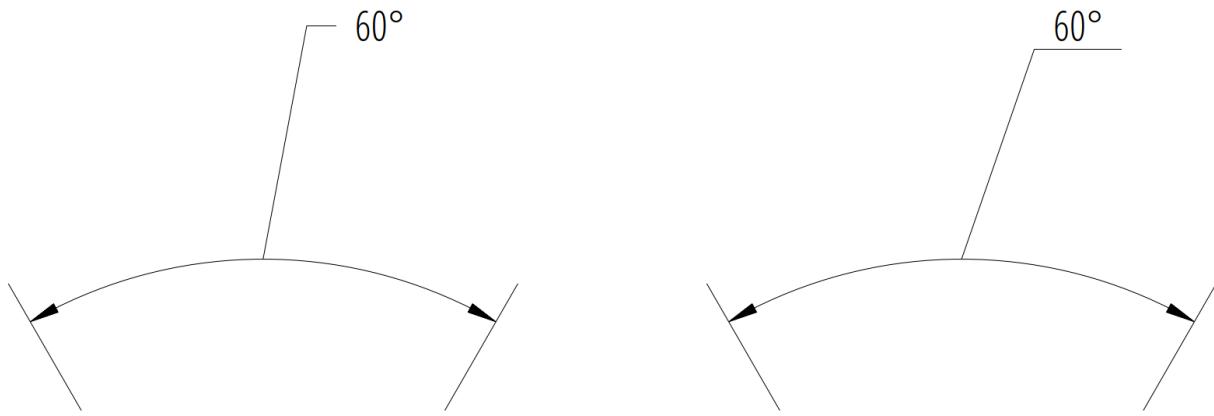
```
dim = msp.add_angular_dim_cra(  
    center=(3, 3),  
    radius=3,  
    distance=1,  
    start_angle=60,  
    end_angle=120,  
)  
dim.set_location((1, 2), relative=True)  
dim.render()
```



Adding a Leader

The method `set_location()` has the option to add a leader line to the measurement text. This also aligns the text rotation to the render UCS x-axis, this means in the default case the measurement text is horizontal. The leader line can be “below” the text or start at the “left” or “right” center of the text, this location is defined by the `dimtad` attribute, 0 means “center” and any value != 0 means “below”.

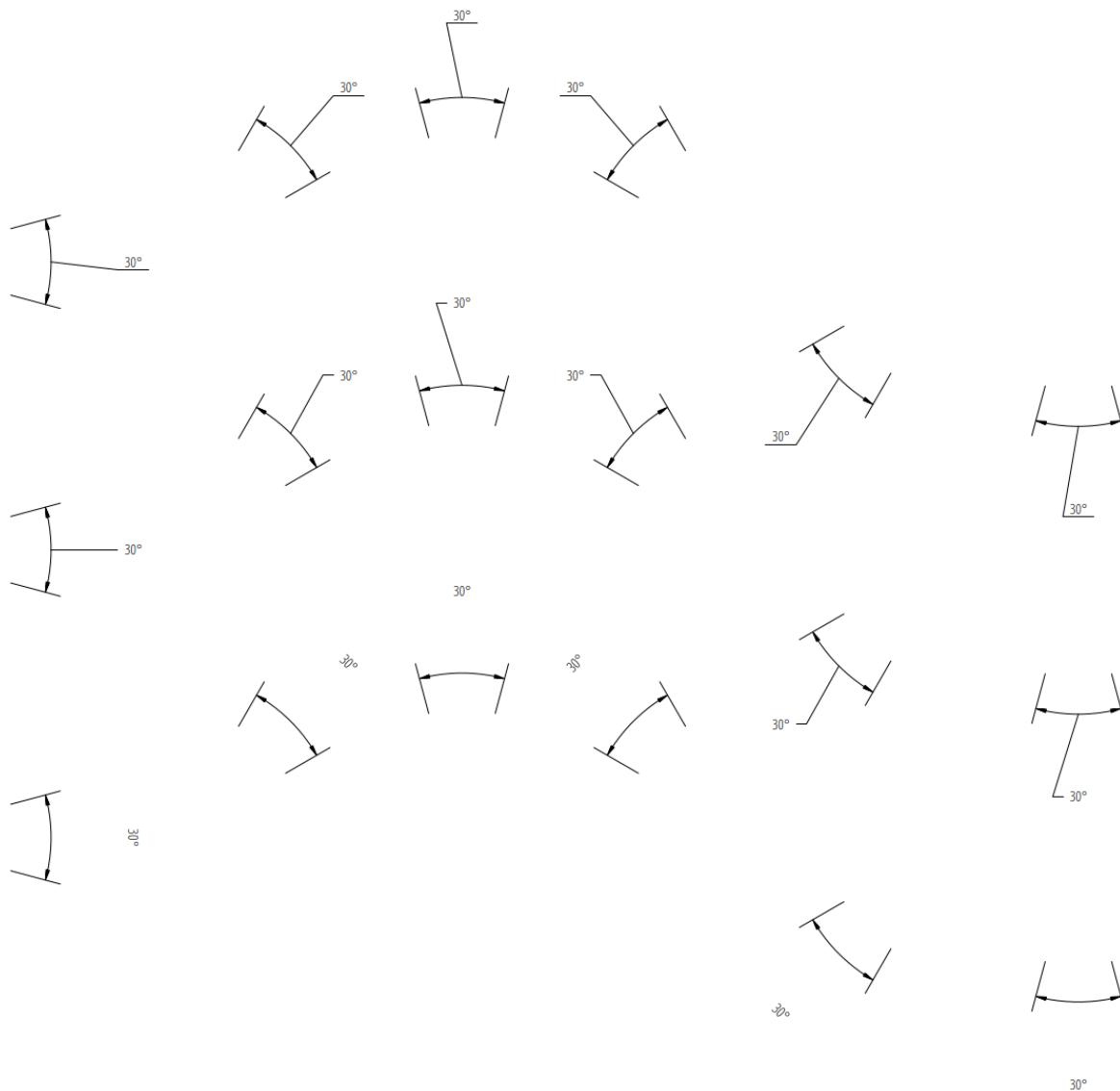
```
for dimtad, x in [(0, 0), (4, 6)]:
    dim = msp.add_angular_dim_cra(
        center=(3 + x, 3),
        radius=3,
        distance=1,
        start_angle=60,
        end_angle=120,
        override={"dimtad": dimtad} # "center" == 0; "below" != 0;
    )
    dim.set_location((1, 2), relative=True, leader=True)
    dim.render()
```



Advanced version which calculates the relative text location: The user location vector has a length 2 and the orientation is defined by *center_angle* pointing away from the center of the angle.

```
import ezdxf
from ezdxf.math import Vec3

doc = ezdxf.new(setup=True)
msp = doc.modelspace()
for dimtad, y, leader in [
    [0, 0, False],
    [0, 7, True],
    [4, 14, True],
]:
    for x, center_angle in [
        (0, 0), (7, 45), (14, 90), (21, 135), (26, 225), (29, 270)
    ]:
        dim = msp.add_angular_dim_cra(
            center=(x, y),
            radius=3.0,
            distance=1.0,
            start_angle=center_angle - 15.0,
            end_angle=center_angle + 15.0,
            override={"dimtad": dimtad},
        )
        # The user location is relative to the center of the dimension line:
        usr_location = Vec3.from_deg_angle(angle=center_angle, length=2.0)
        dim.set_location(usr_location, leader=leader, relative=True)
        dim.render()
```



Overriding Text Rotation

All factory methods supporting the argument `text_rotation` can override the measurement text rotation. The user defined rotation is relative to the render UCS x-axis (default is WCS).

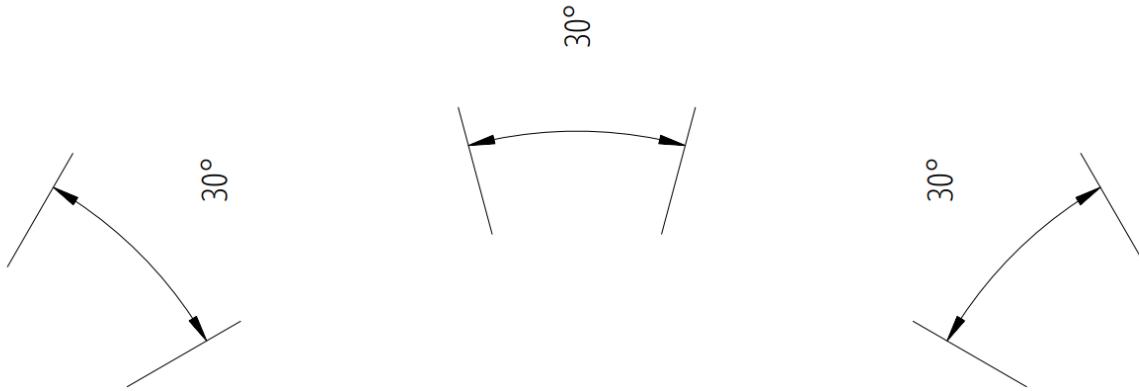
This example uses a relative text location without a leader and forces the text rotation to 90 degrees:

```
for x, center_angle in [(7, 45), (14, 90), (21, 135)]:
    dim = msp.add_angular_dim_cra(
        center=(x, 0),
        radius=3.0,
        distance=1.0,
        start_angle=center_angle - 15.0,
        end_angle=center_angle + 15.0,
        text_rotation=90, # vertical text
    )
```

(continues on next page)

(continued from previous page)

```
usr_location = Vec3.from_deg_angle(angle=center_angle, length=1.0)
dim.set_location(usr_location, leader=False, relative=True)
dim.render()
```



Angular Units

Angular units are set by `dimaunit`:

0	Decimal degrees
1	Degrees/Minutes/Seconds, <code>dimadec</code> controls the shown precision <ul style="list-style-type: none"> • <code>dimadec=0: 30°</code> • <code>dimadec=2: 30°35'</code> • <code>dimadec=4: 30°35'25"</code> • <code>dimadec=7: 30°35'25.15"</code>
2	Grad
3	Radians

```
d1 = 15
d2 = 15.59031944
for x, (dimaunit, dimadec) in enumerate(
    [
        (0, 4),
        (1, 7),
        (2, 4),
        (3, 4),
    ]
):
    dim = msp.add_angular_dim_cra(
        center=(x * 4.0, 0.0),
        radius=3.0,
        distance=1.0,
        start_angle=90.0 - d1,
```

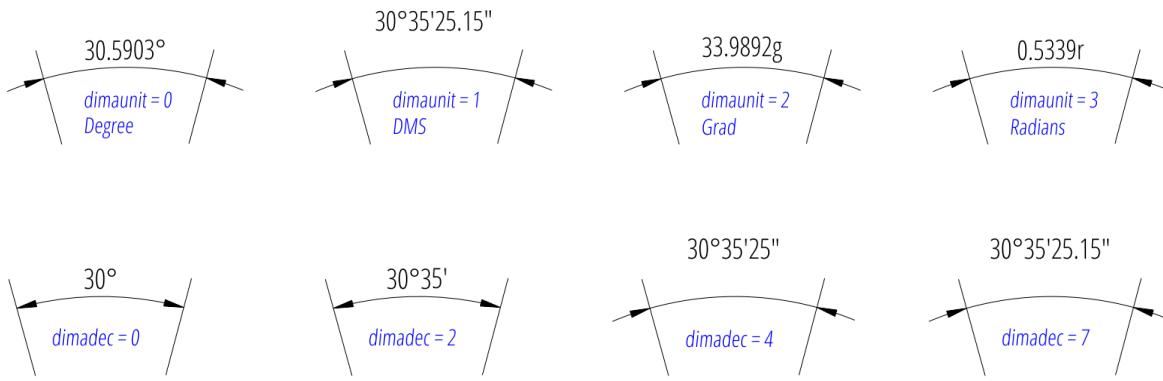
(continues on next page)

(continued from previous page)

```

        end_angle=90.0 + d2,
        override={
            "dimauunit": dimauunit,
            "dimadec": dimadec,
        },
    )
dim.render()

```



Overriding Measurement Text

See Linear Dimension Tutorial: [Overriding Measurement Text](#)

Measurement Text Formatting and Styling

See Linear Dimension Tutorial: [Measurement Text Formatting and Styling](#)

Tolerances and Limits

See Linear Dimension Tutorial: [Tolerances and Limits](#)

6.5.23 Tutorial for Arc Dimensions

Please read the [Tutorial for Linear Dimensions](#) before, if you haven't. This is a repetition of the [Tutorial for Angular Dimensions](#), because `ezdxf` reuses the angular dimension to render arc dimensions. This approach is very different to CAD applications, but also much less work.

Note: `Ezdx`f does not render the arc dimension like CAD applications and does not consider all DIMSTYLE variables, so the rendering results are **very** different from CAD applications.

Dimension Style “EZ_CURVED”

All factory methods to create arc dimensions uses the dimension style “EZ_CURVED” for curved dimension lines which is defined as:

- angle unit is decimal degrees, dimaunit = 0
- measurement text height = 0.25 (drawing scale = 1:100)
- measurement text location is above the dimension line
- closed filled arrow and arrow size dimasz = 0.25
- dimzin = 2, suppresses trailing zeros (e.g. 12.5000 becomes 12.5)
- dimarcsty = 2, disables the arc symbol, 0 renders only an open round bracket "(" in front of the text and 1 for arc symbol above the text is not supported, renders like disabled

For more information go to: [Dimension Style “EZ_CURVED”](#)

Factory Methods to Create Arc Dimensions

Defined by Center, Radius and Angles

The first example shows an arc dimension defined by the center point, radius, start- and end angles:

```
import ezdxf

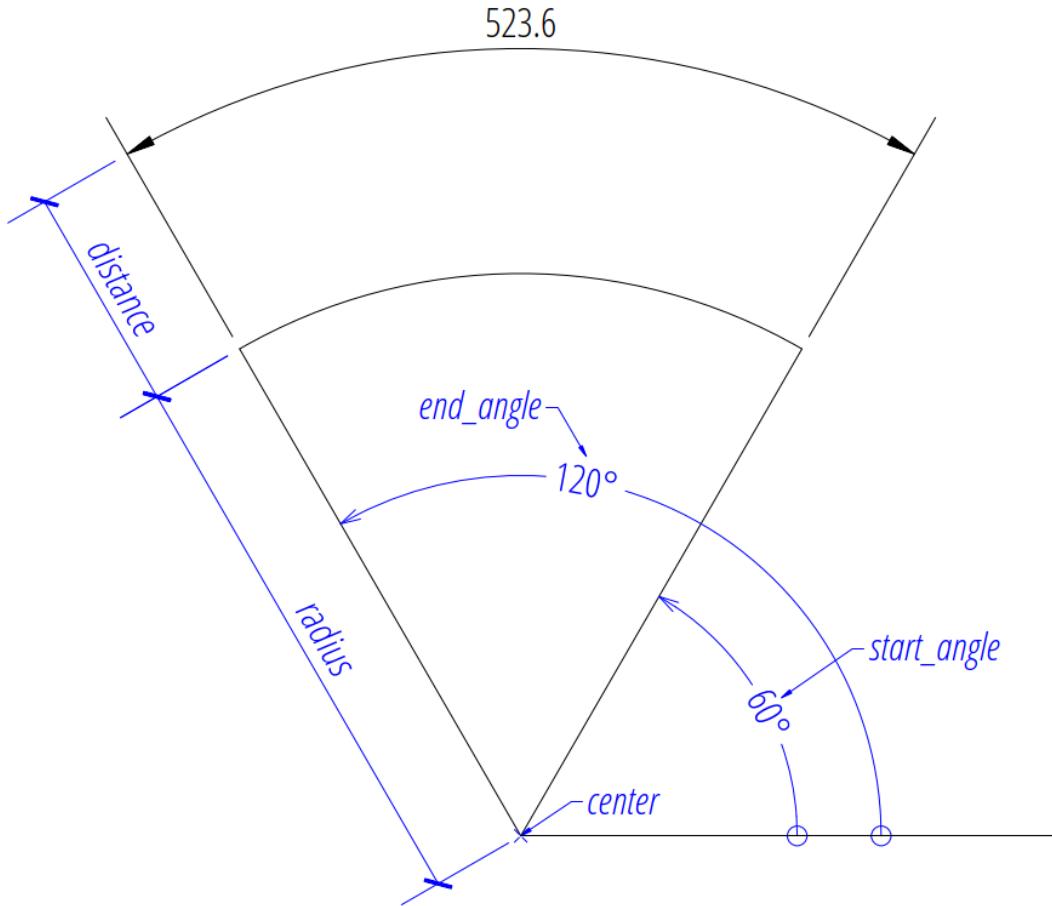
# Use argument setup=True to setup the default dimension styles.
doc = ezdxf.new(setup=True)

# Add new entities to the modelspace:
msp = doc.modelspace()

# Add an arc DIMENSION defined by the center point, start- and end angles,
# the measurement text is placed at the default location above the dimension
# line:
dim = msp.add_arc_dim_cra(
    center=(5, 5), # center point of the angle
    radius=5, # distance from center point to the start of the extension lines
    start_angle=60, # start angle in degrees
    end_angle=120, # end angle in degrees
    distance=2, # distance from start of the extension lines to the dimension line
    dimstyle="EZ_CURVED", # default angular dimension style
)

# Necessary second step to create the BLOCK entity with the dimension geometry.
# Additional processing of the DIMENSION entity could happen between adding
# the entity and the rendering call.
dim.render()
doc.saveas("arc_dimension_cra.dxf")
```

The return value `dim` is **not** a dimension entity, instead a `DimStyleOverride` object is returned, the dimension entity is stored as `dim.dimension`.



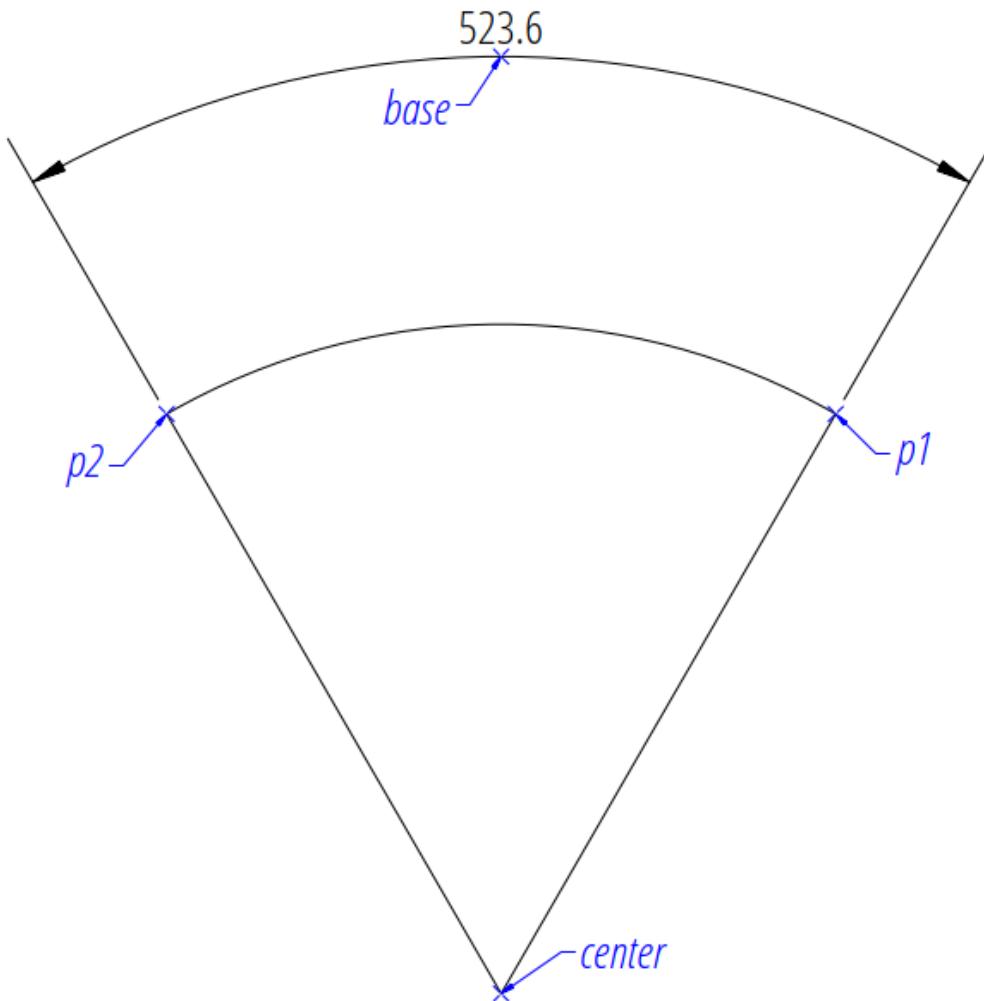
Arc by 3 Points

The next example shows an angular dimension defined by three points, a center point and the two end points of the angle legs, the first point defines the radius, the second point defines only the end angle, the distance from the center point is not relevant:

```
import ezdxf

doc = ezdxf.new(setup=True)
msp = doc.modelspace()

msp.add_arc_dim_3p(
    base=(0, 7), # location of the dimension line
    center=(0, 0), # center point
    p1=(2.5, 4.330127018922193), # 1st point of arc defines start angle and radius
    p2=(-2.5, 4.330127018922194), # 2nd point defines the end angle
).render()
```



Angle from ConstructionArc

The `ezdxf.math.ConstructionArc` provides various class methods for creating arcs and the construction tool can be created from an ARC entity.

Add an angular dimension to an ARC entity:

```
import ezdxf

doc = ezdxf.new(setup=True)
msp = doc.modelspace()

arc = msp.add_arc(
    center=(0, 0),
    radius=5,
    start_angle = 60,
    end_angle = 120,
```

(continues on next page)

(continued from previous page)

```

)
msp.add_arc_dim_arc(
    arc.construction_tool(),
    distance=2,
).render()

```

Placing Measurement Text

The default location of the measurement text depends on various *DimStyle* parameters and is applied if no user defined text location is defined.

Note: Not all possible features of DIMSTYLE are supported by the *ezdxf* rendering procedure and especially for the arc dimension there are less features implemented than for the linear dimension because of the lack of good documentation. If the arc symbol is enabled (*dimarc sym* = 0) only an open round bracket “(“ is rendered in front of the measurement text!

See also:

- Graphical reference of many DIMVARS and some advanced information: *DIMSTYLE Table*
- Source code file `standards.py` shows how to create your own DIMSTYLES.
- The Script `dimension_arc.py` shows examples for angular dimensions.

Default Text Locations

The DIMSTYLE “EZ_CURVED” places the measurement text in the center of the angle above the dimension line. The first examples above show the measurement text at the default text location.

The text direction angle is always perpendicular to the line from the text center to the center point of the angle unless this angle is manually overridden.

Arrows and measurement text are placed “outside” automatically if the available space between the extension lines isn’t sufficient.

For more information go to: *Default Text Locations*

Shift Text From Default Location

The method `shift_text()` shifts the measurement text away from the default location. The shifting direction is aligned to the text rotation of the default measurement text.

For more information go to: *Shift Text From Default Location*

User Defined Text Locations

Beside the default location it is always possible to override the text location by a user defined location.

The coordinates of user locations are located in the rendering UCS and the default rendering UCS is the *WCS*.

For more information go to: *User Defined Text Locations*

Absolute User Location

Absolute placing of the measurement text means relative to the origin of the render UCS.

For more information go to: [User Defined Text Locations](#)

Relative User Location

Relative placing of the measurement text means relative to the middle of the dimension line.

For more information go to: [User Defined Text Locations](#)

Adding a Leader

Add a leader line to the measurement text and set the text rotation to “horizontal”.

For more information go to: [User Defined Text Locations](#)

Overriding Text Rotation

All factory methods supporting the argument `text_rotation` can override the measurement text rotation. The user defined rotation is relative to the render UCS x-axis (default is WCS).

For more information go to: [User Defined Text Locations](#)

Overriding Measurement Text

See Linear Dimension Tutorial: [Overriding Text Rotation](#)

Measurement Text Formatting and Styling

See Linear Dimension Tutorial: [Measurement Text Formatting and Styling](#)

Tolerances and Limits

See Linear Dimension Tutorial: [Tolerances and Limits](#)

6.5.24 Tutorial for Ordinate Dimensions

Please read the [Tutorial for Linear Dimensions](#) before, if you haven't.

Note: *Ezdx*f does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Local Coordinate System

Ordinate dimensioning is used when the x- and the y-coordinates from a location (feature), are the only dimensions necessary. The dimensions to each feature, originate from one datum location, called “origin” in this tutorial.

The local coordinate system (LCS) in which the measurement is done, is defined by the *origin* and the *rotation* angle around the z-axis in the rendering UCS, which is the *WCS* by default.

Factory Methods to Create Ordinate Dimensions

All factory methods for creating ordinate dimensions expect global coordinates to define the feature location.

Global Feature Location

The first example shows ordinate dimensions defined in the render UCS, in this example the *WCS*, this is how the *DIMENSION* entity expects the coordinates of the feature location:

```
import ezdxf
from ezdxf.math import Vec3
from ezdxf.render import forms

# Use argument setup=True to setup the default dimension styles.
doc = ezdxf.new(setup=True)

# Add new entities to the modelspace:
msp = doc.modelspace()
# Add a rectangle: width=4, height = 2.5, lower left corner is WCS(x=2, y=3)
origin = Vec3(2, 3)
msp.add_lwpolyline(
    forms.translate(forms.box(4, 2.5), origin),
    close=True
)

# Add an x-type ordinate DIMENSION with global feature locations:
msp.add_ordinate_x_dim(
    # lower left corner
    feature_location=origin + (0, 0), # feature location in the WCS
    offset=(0, -2), # end of leader, relative to the feature location
    origin=origin,
).render()

msp.add_ordinate_x_dim(
    # lower right corner
    feature_location=origin + (4, 0), # feature location in the WCS
    offset=(0, -2),
    origin=origin,
).render()

# Add an y-type ordinate DIMENSION with global feature locations:
msp.add_ordinate_y_dim(
    # lower right corner
    feature_location=origin + (4, 0), # feature location in the WCS
    offset=(2, 0),
    origin=origin,
).render()

msp.add_ordinate_y_dim()
```

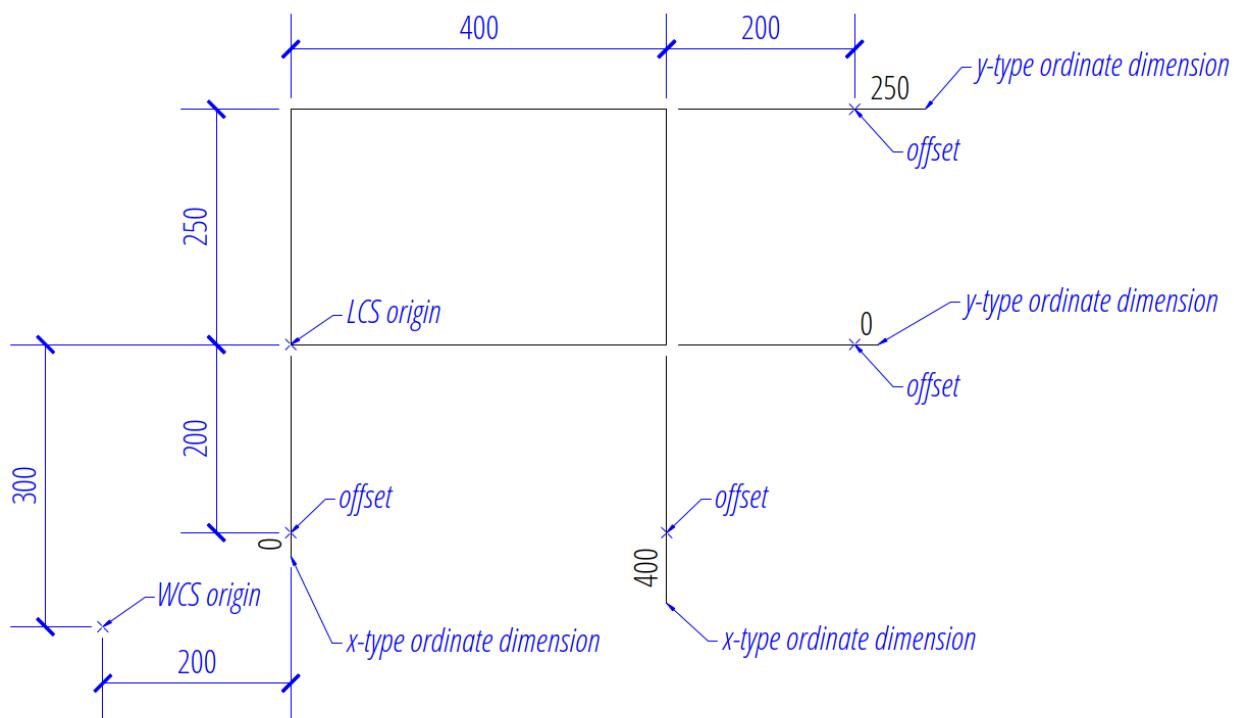
(continues on next page)

(continued from previous page)

```
# upper right corner
feature_location=origin + (4, 2.5), # feature location in the WCS
offset=(2, 0),
origin=origin,
).render()

# Necessary second step to create the BLOCK entity with the dimension geometry.
# Additional processing of the DIMENSION entity could happen between adding
# the entity and the rendering call.
doc.saveas("ord_global_features.dxf")
```

The return value `dim` is **not** a dimension entity, instead a `DimStyleOverride` object is returned, the dimension entity is stored as `dim.dimension`.



Local Feature Location

The previous examples shows that the calculation of the global feature location is cumbersome and it gets even more complicated for a rotated LCS.

This example shows how to use a render `UCS` for using locale coordinates to define the feature locations:

```
import ezdxf
from ezdxf.math import Vec3, UCS
from ezdxf.render import forms

doc = ezdxf.new(setup=True)
msp = doc.modelspace()
```

(continues on next page)

(continued from previous page)

```

# Create a special DIMSTYLE for "vertical" centered measurement text:
dimstyle = doc.dimstyles.duplicate_entry("EZDXF", "ORD_CENTER")
dimstyle.dxf.dimtad = 0 # "vertical" centered measurement text

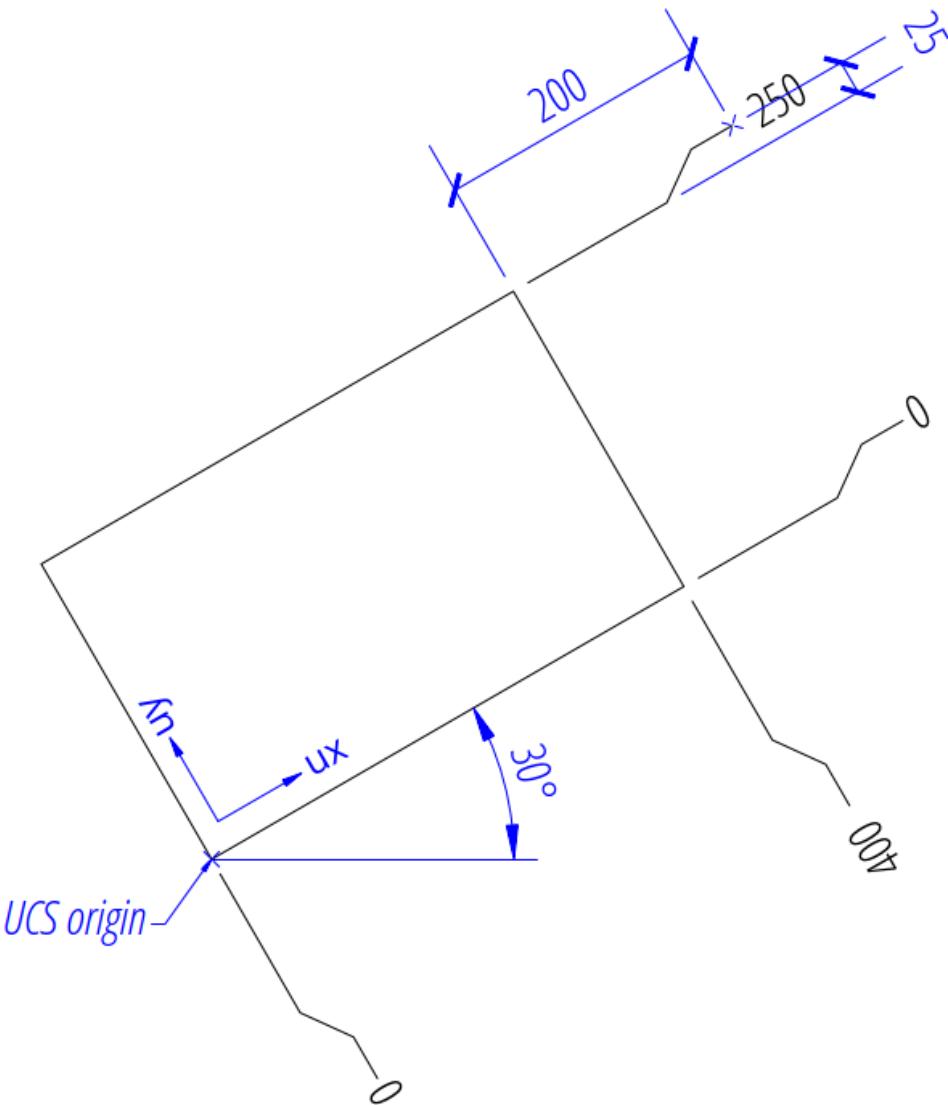
# Add a rectangle: width=4, height = 2.5, lower left corner is WCS(x=2, y=3),
# rotated about 30 degrees:
origin = Vec3(2, 3)
msp.add_lwpolyline(
    forms.translate(forms.rotate(forms.box(4, 2.5), 30), origin),
    close=True
)

# Define the rotated local render UCS.
# The origin is the lower-left corner of the rectangle and the axis are
# aligned to the rectangle edges:
# The y-axis "uy" is calculated automatically by the right-hand rule.
ucs = UCS(origin, ux=Vec3.from_deg_angle(30), uz=(0, 0, 1))

# Add a x-type ordinate DIMENSION with local feature locations:
# the origin is now the origin of the UCS, which is (0, 0) the default value of
# "origin" and the feature coordinates are located in the UCS:
msp.add_ordinate_x_dim(
    # lower left corner
    feature_location=(0, 0), # feature location in the UCS
    offset=(0.25, -2), # leader with a "knee"
    dimstyle="ORD_CENTER",
).render(ucs=ucs) # Important when using a render UCS!
msp.add_ordinate_x_dim(
    # lower right corner
    feature_location=(4, 0), # feature location in the UCS
    offset=(0.25, -2), # leader with a "knee"
    dimstyle="ORD_CENTER",
).render(ucs=ucs) # Important when using a render UCS!

# Add a y-type ordinate DIMENSION with local feature coordinates:
msp.add_ordinate_y_dim(
    # lower right corner
    feature_location=(4, 0), # feature location in the UCS
    offset=(2, 0.25), # leader with a "knee"
    dimstyle="ORD_CENTER",
).render(ucs=ucs) # Important when using a render UCS!
msp.add_ordinate_y_dim(
    # upper right corner
    feature_location=(4, 2.5), # feature location in the UCS
    offset=(2, 0.25), # leader with a "knee"
    dimstyle="ORD_CENTER",
).render(ucs=ucs) # Important when using a render UCS!
doc.saveas("ord_local_features.dxf")

```



Placing Measurement Text

The *ezdxf* ordinate DIMENSION renderer places the measurement text always at the default location, because the location of the leader end point is given by the argument *offset* in the factory methods, which provides a flexible way to place the measurement text, overriding the text location by an explicit user location is not supported, also the user text rotation is not supported, the text is always aligned to the local coordinate system x- and y-axis.

See also:

- Graphical reference of many DIMVARS and some advanced information: [DIMSTYLE Table](#)
- Source code file `standards.py` shows how to create your own DIMSTYLES.
- The Script `dimension_ordinate.py` shows examples for angular dimensions.

Overriding Measurement Text

See Linear Dimension Tutorial: [Overriding Text Rotation](#)

Measurement Text Formatting and Styling

See Linear Dimension Tutorial: [Measurement Text Formatting and Styling](#)

Tolerances and Limits

See Linear Dimension Tutorial: [Tolerances and Limits](#)

6.5.25 Tutorial for the Geo Add-on

This tutorial shows how to load a GPS track into a geo located DXF file and also the inverse operation, exporting geo located DXF entities as GeoJSON files.

Please read the section [Intended Usage](#) in the documentation of the `ezdxf.addons.geo` module first.

Warning: TO ALL BEGINNERS!

If you are just learning to work with geospatial data, using DXF files is not the way to go! DXF is not the first choice for storing data for spatial data analysts. If you run into problems I cannot help you as I am just learning myself.

The complete source code and test data for this tutorial are available in the github repository:

<https://github.com/mozman/ezdxf/tree/master/docs/source/tutorials/src/geo>

Setup Geo Location Reference

The first step is setting up the geo location reference, which is **not** doable with ezdxf yet - this feature may come in the future - but for now you have to use a CAD application to do this. If the DXF file has no geo location reference the projected 2D coordinates are most likely far away from the WCS origin (0, 0), use the CAD command “ZOOM EXTENDS” to find the data.

Load GPX Data

The GPX format stores GPS data in a XML format, use the `ElementTree` class to load the data:

```
def load_gpx_track(p: Path) -> Iterable[Tuple[float, float]]:
    """Load all track points from all track segments at once."""
    gpx = ET.parse(p)
    root = gpx.getroot()
    for track_point in root.findall("./gpx:trkpt", GPX_NS):
        data = track_point.attrib
        # Elevation is not supported by the geo add-on.
        yield float(data["lon"]), float(data["lat"])
```

The loaded GPS data has a WSG84 EPSG:4326 projection as longitude and latitude in decimal degrees. The next step is to create a `GeoProxy` object from this data, the `GeoProxy.parse()` method accepts a `__geo_interface__` mapping or a Python object with a `__geo_interface__` attribute/property. In this case as simple “`LineString`” object for all GPS points is sufficient:

```
def add_gpx_track(msp, track_data, layer: str):
    geo_mapping = {
        "type": "LineString",
        "coordinates": track_data,
    }
    geo_track = geo.GeoProxy.parse(geo_mapping)
```

Transform the data from the polar representation EPSG:4326 into a 2D cartesian map representation EPSG:3395 called “World Mercator”, this is the only projection supported by the add-on, without the need to write a custom transformation function:

```
geo_track.globe_to_map()
```

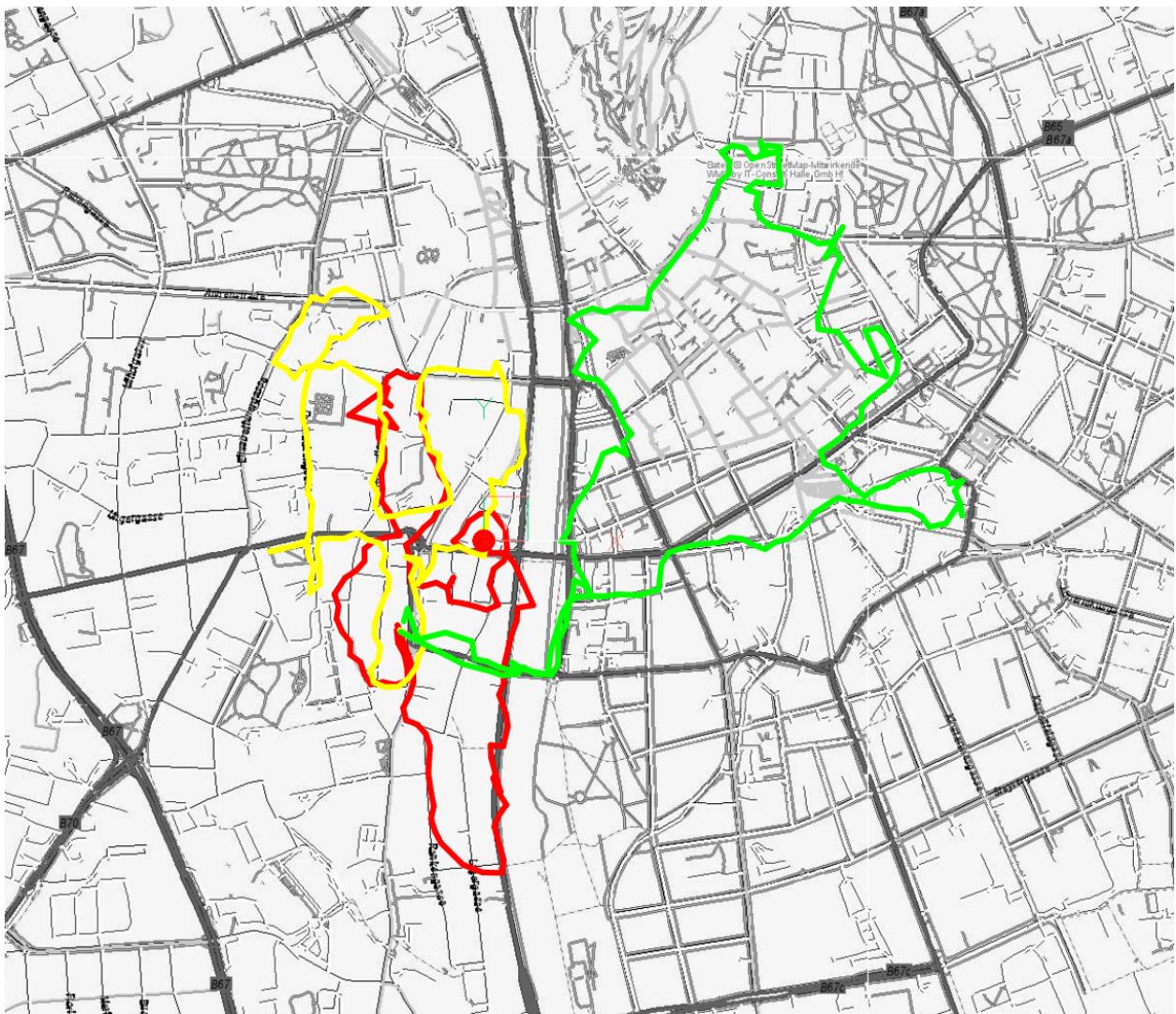
The data is now transformed into 2D cartesian coordinates in meters and most likely far away from origin (0, 0), the data stored in the GEODATA entity helps to transform the data into the DXF WCS in modelspace units, if the DXF file has no geo location reference you have to stick with the large coordinates:

```
# Load geo data information from the DXF file:
geo_data = msp.get_geodata()
if geo_data:
    # Get the transformation matrix and epsg code:
    m, epsg = geo_data.get_crs_transformation()
else:
    # Identity matrix for DXF files without a geo location reference:
    m = Matrix44()
    epsg = 3395
# Check for compatible projection:
if epsg == 3395:
    # Transform CRS coordinates into DXF WCS:
    geo_track.crs_to_wcs(m)
    # Create DXF entities (LWPOLYLINE)
    for entity in geo_track.to_dxf_entities(dxfattribs={"layer": layer}):
        # Add entity to the modelspace:
        msp.add_entity(entity)
else:
    print(f"Incompatible CRS EPSG: {epsg}")
```

We are ready to save the final DXF file:

```
doc.saveas(str(out_path))
```

In BricsCAD the result looks like this, the underlying images were added by the BricsCAD command MAPCONNECT and such a feature is **not** planned for the add-on:



Export DXF Entities as GeoJSON

This will only work with a proper geo location reference, the code shown accepts also WCS data from DXF files without a GEODATA object, but the result is just unusable - but in valid GeoJSON notation.

First get epsg code and the CRS transformation matrix:

```
# Get the geo location information from the DXF file:
geo_data = msp.get_geodata()
if geo_data:
    # Get transformation matrix and epsg code:
    m, epsg = geo_data.get_crs_transformation()
else:
    # Identity matrix for DXF files without geo reference data:
    m = Matrix44()
```

Query the DXF entities to export:

```
for track in msp.query("LWPOLYLINE"):
    export_geojson(track, m)
```

Create a GeoProxy object from the DXF entity:

```
def export_geojson(entity, m):
    # Convert DXF entity into a GeoProxy object:
    geo_proxy = geo.proxy(entity)
```

Transform DXF WCS coordinates in modelspace units into the CRS coordinate system by the transformation matrix *m*:

```
# Transform DXF WCS coordinates into CRS coordinates:
geo_proxy.wcs_to_crs(m)
```

The next step assumes a EPSG:3395 projection, everything else needs a custom transformation function:

```
# Transform 2D map projection EPSG:3395 into globe (polar)
# representation EPSG:4326
geo_proxy.map_to_globe()
```

Use the `json` module from the Python standard library to write the GeoJSON data, provided by the `GeoProxy.__geo_interface__` property:

```
# Export GeoJSON data:
name = entity.dxf.layer + ".geojson"
with open(TRACK_DATA / name, "wt", encoding="utf8") as fp:
    json.dump(geo_proxy.__geo_interface__, fp, indent=2)
```

The content of the GeoJSON file looks like this:

```
{
    "type": "LineString",
    "coordinates": [
        [
            [15.430999, 47.06503],
            [15.431039, 47.064797]
        ],
        [
            [15.431206, 47.064582],
            [15.431283, 47.064342]
        ],
        ...
    ]
}
```

Custom Transformation Function

This sections shows how to use the GDAL package to write a custom transformation function. The example reimplements the builtin transformation from unprojected WGS84 coordinates to 2D map coordinates EPSG:3395 “World Mercator”:

```

from osgeo import osr
from ezdxf.math import Vec3

# GPS track in WGS84, load_gpx_track() code see above
gpx_points = list(load_gpx_track('track1.gpx'))

# Create source coordinate system:
src_datum = osr.SpatialReference()
src_datum.SetWellKnownGeoCS('WGS84')

# Create target coordinate system:
target_datum = osr.SpatialReference()
target_datum.SetWellKnownGeoCS('EPSG:3395')

# Create transformation object:
ct = osr.CoordinateTransform(src_datum, target_datum)

# Create GeoProxy() object:
geo_proxy = GeoProxy.parse({
    'type': 'LineString',
    'coordinates': gpx_points
})

# Apply a custom transformation function to all coordinates:
geo_proxy.apply(lambda v: Vec3(ct.TransformPoint(v.x, v.y)))

```

The same example with the pyproj package:

```

from pyproj import Transformer
from ezdxf.math import Vec3

# GPS track in WGS84, load_gpx_track() code see above
gpx_points = list(load_gpx_track('track1.gpx'))

# Create transformation object:
ct = Transformer.from_crs('EPSG:4326', 'EPSG:3395')

# Create GeoProxy() object:
geo_proxy = GeoProxy.parse({
    'type': 'LineString',
    'coordinates': gpx_points
})

# Apply a custom transformation function to all coordinates:
geo_proxy.apply(lambda v: Vec3(ct.transform(v.x, v.y)))

```

Polygon Validation by Shapely

Ezdx tries to avoid to create invalid polygons from HATCH entities like a hole in another hole, but not all problems are detected by ezdx, especially overlapping polygons. For a reliable and robust result use the Shapely package to check for valid polygons:

```

import ezdxf
from ezdxf.addons import geo
from shapely.geometry import shape

```

(continues on next page)

(continued from previous page)

```
# Load DXF document including HATCH entities.
doc = ezdxf.readfile('hatch.dxf')
msp = doc.modelspace()

# Test a single entity
# Get the first DXF hatch entity:
hatch_entity = msp.query('HATCH').first

# Create GeoProxy() object:
hatch_proxy = geo.proxy(hatch_entity)

# Shapely supports the __geo_interface__
shapely_polygon = shape(hatch_proxy)

if shapely_polygon.is_valid:
    ...
else:
    print(f'Invalid Polygon from {str(hatch_entity)} .')

# Remove invalid entities by a filter function
def validate(geo_proxy: geo.GeoProxy) -> bool:
    # Multi-entities are divided into single entities:
    # e.g. MultiPolygon is verified as multiple single Polygon entities.
    if geo_proxy.geotype == 'Polygon':
        return shape(geo_proxy).is_valid
    return True

# The gfilter() function let only pass compatible DXF entities
msp_proxy = geo.GeoProxy.from_dxf_entities(geo.gfilter(msp))

# remove all mappings for which validate() returns False
msp_proxy.filter(validate)
```

Interface to GDAL/OGR

The GDAL/OGR package has no direct support for the `__geo_interface__`, but has builtin support for the GeoJSON format:

```
from osgeo import ogr
from ezdxf.addons import geo
from ezdxf.render import random_2d_path
import json

p = geo.GeoProxy({'type': 'LineString', 'coordinates': list(random_2d_path(20))})
# Create a GeoJSON string from the __geo_interface__ object by the json
# module and feed the result into ogr:
line_string = ogr.CreateGeometryFromJson(json.dumps(p.__geo_interface__))

# Parse the GeoJSON string from ogr by the json module and feed the result
# into a GeoProxy() object:
p2 = geo.GeoProxy.parse(json.loads(line_string.ExportToJson()))
```

6.5.26 Storing Custom Data in DXF Files

This tutorial describes how to store custom data in DXF files using standard DXF features.

Saving data in comments is not covered in this section, because comments are not a reliable way to store information in DXF files and *ezdxf* does not support adding comments to DXF files. Comments are also ignored by *ezdxf* and many other DXF libraries when loading DXF files, but there is a *ezdxf.comments* module to load comments from DXF files.

The DXF data format is a very versatile and flexible data format and supports various ways to store custom data. This starts by setting special header variables, storing XData, AppData and extension dictionaries in DXF entities and objects, storing XRecords in the OBJECTS section and ends by using proxy entities or even extending the DXF format by user defined entities and objects.

This is the common prolog for all Python code examples shown in this tutorial:

```
import ezdxf
doc = ezdxf.new()
msp = doc.modelspace()
```

Retrieving User Data

Retrieving the is a simple task by *ezdxf*, but often not possible in CAD applications without using the scripting features (AutoLISP) or even the SDK.

AutoLISP Resources

- Autodesk Developer Documentation
- AfraLISP
- Lee Mac Programming

Warning: I have no experience with AutoLISP so far and I created this scripts for AutoLISP while writing this tutorial. There may be better ways to accomplish these tasks, and feedback on this is very welcome. Everything is tested with BricsCAD and should also work with the full version of AutoCAD.

Header Section

The HEADER section has tow ways to store custom data.

Predefined User Variables

There are ten predefined user variables, five 16-bit integer variables called \$USERI1 up to \$USERI5 and five floating point variables (reals) called \$USERR1 up to \$USERR5. This is very limited and the data maybe will be overwritten by the next application which opens and saves the DXF file. Advantage of this methods is, it works for all supported DXF versions starting at R12.

Settings the data:

```
doc.header["$USERI1"] = 4711
doc.header["$USERR1"] = 3.141592
```

Getting the data by *ezdxf*:

```
i1 = doc.header["$USERI1"]
r1 = doc.header["$USERR1"]
```

Getting the data in *BricsCAD* at the command line:

```
: USERI1
New current value for USERI1 (-32768 to 32767) <4711>:
```

Getting the data by AutoLISP:

```
: (getvar 'USERI1)
4711
```

Setting the value by AutoLISP:

```
: (setvar 'USERI1 1234)
1234
```

Custom Document Properties

This method defines custom document properties, but requires at least DXF R2004. The custom document properties are stored in a *CustomVars* instance in the *custom_vars* attribute of the *HeaderSection* object and supports only string values.

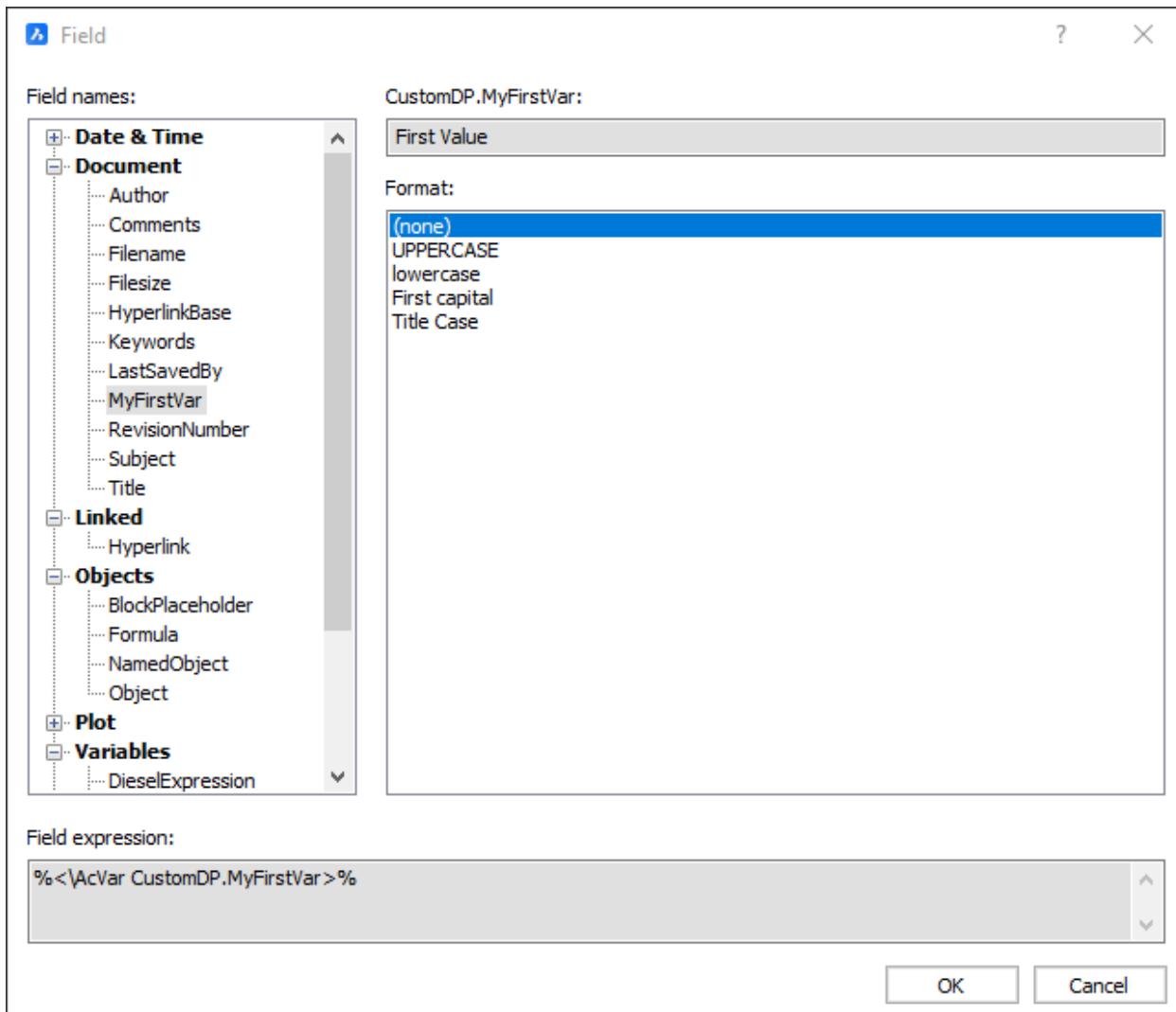
Settings the data:

```
doc.header.custom_vars.append("MyFirstVar", "First Value")
```

Getting the data by *ezdxf*:

```
my_first_var = doc.header.custom_vars.get("MyFirstVar", "Default Value")
```

The document property MyFirstVar is available in *BricsCAD* as FIELD variable:



AutoLISP script for getting the custom document properties:

```
(defun C:CUSTOMDOCPROPS (/ Info Num Index Custom)
  (vl-load-com)
  (setq acadObject (vlax-get-acad-object))
  (setq acadDocument (vla-get-ActiveDocument acadObject))

  ;;Get the SummaryInfo
  (setq Info (vlax-get-Property acadDocument 'SummaryInfo))
  (setq Num (vla-NumCustomInfo Info))
  (setq Index 0)
  (repeat Num
    (vla-getCustomByIndex Info Index 'ID 'Value)
    (setq Custom (cons (cons ID Value) Custom)))
    (setq Index (1+ Index)))
  ) ;repeat

  (if Custom (reverse Custom))
)
```

Running the script in BricsCAD:

```
: (load "customdocprops.lsp")
C:CUSTOMDOCPROPS
: CUSTOMDOCPROPS
(("MyFirstVar" . "First Value"))
```

Meta Data

Starting with version v0.16.4 *ezdxf* stores some meta data in the DXF file and the AppID EZDXF will be created. Two entries will be added to the *MetaData* instance, the CREATED_BY_EZDXF for DXF documents created by *ezdxf* and the entry WRITTEN_BY_EZDXF if the DXF document will be saved by *ezdxf*. The marker string looks like this "0.17b0 @ 2021-09-18T05:14:37.921826+00:00" and contains the *ezdxf* version and an UTC timestamp in ISO format.

You can add your own data to the *MetaData* instance as a string with a maximum of 254 characters and choose a good name which may never be used by *ezdxf* in the future.

```
metadata = doc.ezdxf_metadata()
metadata["MY_UNIQUE_KEY"] = "my additional meta data"

print(metadata.get("CREATED_BY_EZDXF"))
print(metadata.get("MY_UNIQUE_KEY"))
```

The data is stored as XDATA in then BLOCK entity of the model space for DXF R12 and for DXF R2000 and later as a DXF *Dictionary* in the root dictionary by the key EZDXF_META. See following chapters for accessing such data by AutoLISP.

XDATA

Extended Data (XDATA) is a way to attach arbitrary data to DXF entities. Each application needs a unique AppID registered in the AppID table to add XDATA to an entity. The AppID ACAD is reserved and by using *ezdxf* the AppID EZDXF is also registered automatically. The total size of XDATA for a single DXF entity is limited to 16kB for AutoCAD. XDATA is supported by all DXF versions and is accessible by AutoLISP.

The valid group codes for extended data are limited to the following values, see also the internals of *Extended Data*:

Group Code	Description
1000	Strings up to 255 bytes long
1001	(fixed) Registered application name up to 31 bytes long
1002	(fixed) An extended data control string '{' or '}'
1004	Binary data
1005	Database Handle of entities in the drawing database
1010	3D point, in the order X, Y, Z that will not be modified at any transformation of the entity
1011	A WCS point that is moved, scaled, rotated and mirrored along with the entity
1012	A WCS displacement that is scaled, rotated and mirrored along with the entity, but not moved
1013	A WCS direction that is rotated and mirrored along with the entity but not moved and scaled.
1040	A real value
1041	Distance, a real value that is scaled along with the entity
1042	Scale Factor, a real value that is scaled along with the entity
1070	A 16-bit integer (signed or unsigned)
1071	A 32-bit signed (long) integer

Group codes are not unique in the XDATA section and can be repeated, therefore tag order matters.

```

# register your appid
APPID = "YOUR_UNIQUE_ID"
doc.appids.add(APPID)

# create a DXF entity
line = msp.add_line((0, 0), (1, 0))

# setting the data
line.set_xdata(APPID, [
    # basic types
    (1000, "custom text"),
    (1040, 3.141592),
    (1070, 4711), # 16bit
    (1071, 1_048_576), # 32bit
    # points and vectors
    (1010, (10, 20, 30)),
    (1011, (11, 21, 31)),
    (1012, (12, 22, 32)),
    (1013, (13, 23, 33)),
    # scaled distances and factors
    (1041, 10),
    (1042, 10),
])

# getting the data
if line.has_xdata(APPID):
    tags = line.get_xdata(APPID)
    print(f"{str(line)} has {len(tags)} tags of XDATA for AppID {APPID!r}")
    for tag in tags:
        print(tag)

```

AutoLISP script for getting XDATA for AppID YOUR_UNIQUE_ID:

```

(defun C:SHOWXDATA (/ entity_list xdata_list)
  (setq entity_list (entget (car (entsel)) '("YOUR_UNIQUE_ID")))
  (setq xdata_list (assoc -3 entity_list))
  (car (cdr xdata_list)))
)

```

Script output:

```

: SHOWXDATA
Select entity: ("YOUR_UNIQUE_ID" (1000 . "custom text") (1040 . 3.141592) ...

```

See also:

- [AfraLISP XDATA tutorial](#)
- [Extended Data \(XDATA\) Reference](#)

XDATA Helper Classes

The `XDataUserList` and `XDataUserDict` are helper classes to manage XDATA content in a simple way.

Both classes store the Python types `int`, `float` and `str` and the `ezdxf` type `Vec3`. As the names suggests has the `XDataUserList` a list-like interface and the `XDataUserDict` a dict-like interface. This classes can not contain additional container types, but multiple lists and/or dicts can be stored in the same XDATA section for the same AppID.

These helper classes uses a fixed group code for each data type:

1001	strings (max. 255 chars)
1040	floats
1071	32-bit ints
1010	Vec3

Additional required imports for these examples:

```
from eздxf.math import Vec3
from eздxf.entities.xdata import XDataUserDict, XDataUserList
```

Example for *XDataUserDict*:

Each XDataUserDict has a unique name, the default name is “DefaultDict” and the default AppID is EZDXF. If you use your own AppID, don’t forget to create the requited AppID table entry like doc.appids.new("MyAppID"), otherwise AutoCAD will not open the DXF file.

```
doc = eздxf.new()
msp = doc.modelspace()
line = msp.add_line((0, 0), (1, 0))

with XDataUserDict.entity(line) as user_dict:
    user_dict["CreatedBy"] = "mozman"
    user_dict["Float"] = 3.1415
    user_dict["Int"] = 4711
    user_dict["Point"] = Vec3(1, 2, 3)
```

If you modify the content of without using the context manager *entity()*, you have to call *commit()* by yourself, to transfer the modified data back into the XDATA section.

Getting the data back from an entity:

```
with XDataUserDict.entity(line) as user_dict:
    print(user_dict)
    # acts like any other dict()
    storage = dict(user_dict)
```

Example for *XDataUserList*:

This example stores the data in a XDataUserList named “AppendedPoints”, the default name is “DefaultList” and the default AppID is EZDXF.

```
with XDataUserList.entity(line, name="AppendedPoints") as user_list:
    user_list.append(Vec3(1, 0, 0))
    user_list.append(Vec3(0, 1, 0))
    user_list.append(Vec3(0, 0, 1))
```

Now the content of both classes are stored in the same XDATA section for AppID EZDXF. The XDataUserDict is stored by the name “DefaultDict” and the XDataUserList is stored by the name “AppendedPoints”.

Getting the data back from an entity:

```
with XDataUserList.entity(line, name="AppendedPoints") as user_list:
    print(user_list)
    storage = list(user_list)

print(f"Copy of XDataUserList: {storage}")
```

See also:

- [XDataUserList](#) class
- [XDataUserDict](#) class

Extension Dictionaries

Extension dictionaries are another way to attach custom data to any DXF entity. This method requires DXF R13/14 or later. I will use the short term XDICT for extension dictionaries in this tutorial.

The *Extension Dictionary* is a regular DXF [Dictionary](#) which can store (key, value) pairs where the key is a string and the value is a DXF object from the OBJECTS section but not graphical DXF entities. The usual objects to store custom data are [DictionaryVar](#) to store simple strings and [XRecord](#) to store complex data.

Unlike XDATA, custom data attached by extension dictionary will not be transformed along with the DXF entity!

This example shows how to manage the XDICT and to store simple strings as [DictionaryVar](#) objects in the XDICT, to store more complex data go to the next section [XRecord](#).

1. Get or create the XDICT for an entity:

```
# create a DXF entity
line = msp.add_line((0, 0), (1, 0))

if line.has_extension_dict:
    # get the extension dictionary
    xdict = line.get_extension_dict()
else:
    # create a new extension dictionary
    xdict = line.new_extension_dict()
```

2. Add strings as [DictionaryVar](#) objects to the XDICT. No AppIDs required, but existing keys will be overridden, so be careful by choosing your keys:

```
xdict.add_dictionary_var("DATA1", "Your custom data string 1")
xdict.add_dictionary_var("DATA2", "Your custom data string 2")
```

3. Retrieve the strings from the XDICT as [DictionaryVar](#) objects:

```
print(f"DATA1 is '{xdict['DATA1'].value}'")
print(f"DATA2 is '{xdict['DATA2'].value}'")
```

The AutoLISP access to DICTIONARIES is possible, but it gets complex and I'm only referring to the [AfraLISP Dictionaries and XRecords](#) tutorial.

See also:

- [AfraLISP Dictionaries and XRecords Tutorial](#)
- [Extension Dictionary Reference](#)
- [DXF Dictionary Reference](#)
- [DictionaryVar Reference](#)

XRecord

The [XRecord](#) object can store arbitrary data like the XDATA section, but is not limited by size and can use all group codes in the range from 1 to 369 for [DXF Tags](#). The [XRecord](#) can be referenced by any DXF [Dictionary](#), other

XRecord objects (tricky ownership!), the XDATA section (store handle by group code 1005) or any other DXF object by adding the XRecord object to the *Extension Dictionary* of the DXF entity.

It is recommended to follow the DXF reference to assign appropriate group codes to *DXF Tags*. My recommendation is shown in the table below, but all group codes from 1 to 369 are valid. I advice against using the group codes 100 and 102 (structure tags) to avoid confusing generic tag loaders. Unfortunately, Autodesk doesn't like general rules and uses DXF format exceptions everywhere.

1	strings (max. 2049 chars)
2	structure tags as strings like " { " and " } "
10	points and vectors
40	floats
90	integers
330	handles

Group codes are not unique in XRecord and can be repeated, therefore tag order matters.

This example shows how to attach a *XRecord* object to a LINE entity by *Extension Dictionary*:

```
line = msp.add_line((0, 0), (1, 0))
line2 = msp.add_line((0, 2), (1, 2))

if line.has_extension_dict:
    xdict = line.get_extension_dict()
else:
    xdict = line.new_extension_dict()

xrecord = xdict.add_xrecord("DATA1")
xrecord.reset([
    (1, "text1"), # string
    (40, 3.141592), # float
    (90, 256), # 32-bit int
    (10, (1, 2, 0)), # points and vectors
    (330, line2.dxf.handle) # handles
])

print(xrecord.tags)
```

Script output:

```
[DXFTag(1, 'text1'),
 DXFTag(40, 3.141592),
 DXFTag(90, 256),
 DXFVertex(10, (1.0, 2.0, 0.0)),
 DXFTag(330, '30')]
```

Unlike XDATA, custom data attached by extension dictionary will not be transformed along with the DXF entity! To react to entity modifications by a CAD applications it is possible to write event handlers by AutoLISP, see the [AfraLISP Reactors Tutorial](#) for more information. This very advanced stuff!

See also:

- [AfraLISP Dictionaries and XRecords Tutorial](#)
- [AfraLISP Reactors Tutorial](#)
- [XRecord Reference](#)

- helper functions: `ezdxf.lldxf.types.dxftag()` and `ezdxf.lldxf.types.tuples_to_tags()`

XRecord Helper Classes

The `UserRecord` and `BinaryRecord` are helper classes to manage XRECORD content in a simple way. The `UserRecord` manages the data as plain Python types: `dict`, `list`, `int`, `float`, `str` and the `ezdxf` types `Vec2` and `Vec3`. The top level type for the `UserRecord.data` attribute has to be a `list`. The `BinaryRecord` stores arbitrary binary data as `BLOB`. These helper classes uses fixed group codes to manage the data in XRECORD, you have no choice to change them.

Additional required imports for these examples:

```
from pprint import pprint
import eздxf
from eздxf.math import Vec3
from eздxf.urecord import UserRecord, BinaryRecord
from eздxf.entities import XRecord
import zlib
```

Example 1: Store entity specific data in the *Extension Dictionary*:

```
line = msp.add_line((0, 0), (1, 0))
xdict = line.new_extension_dict()
xrecord = xdict.add_xrecord("MyData")

with UserRecord(xrecord) as user_record:
    user_record.data = [ # top level has to be a list!
        "MyString",
        4711,
        3.1415,
        Vec3(1, 2, 3),
        {
            "MyIntList": [1, 2, 3],
            "MyFloatList": [4.5, 5.6, 7.8],
        },
    ]
```

Example 1: Get entity specific data back from the *Extension Dictionary*:

```
if line.has_extension_dict:
    xdict = line.get_extension_dict()
    xrecord = xdict.get("MyData")
    if isinstance(xrecord, XRecord):
        user_record = UserRecord(xrecord)
        pprint(user_record.data)
```

If you modify the content of `UserRecord.data` without using the context manager, you have to call `commit()` by yourself, to store the modified data back into the XRECORD.

Example 2: Store arbitrary data in DICTIONARY objects. The XRECORD is stored in the named DICTIONARY, called `rootdict` in `ezdxf`. This DICTIONARY is the root entity for the tree-like data structure stored in the OBJECTS section, see also the documentation of the `ezdxf.sections.objects` module.

```
# Get the existing DICTIONARY object or create a new DICTIONARY object:
my_dict = doc.objects.rootdict.get_required_dict("MyDict")
```

(continues on next page)

(continued from previous page)

```
# Create a new XRECORD object, the DICTIONARY object is the owner of this
# new XRECORD:
xrecord = my_dict.add_xrecord("MyData")

# This example creates the user record without the context manager.
user_record = UserRecord(xrecord)

# Store user data:
user_record.data = [
    "Just another user record",
    4711,
    3.1415,
]
# Store user data in associated XRECORD:
user_record.commit()
```

Example 2: Get user data back from the DICTIONARY object

```
my_dict = doc.rootdict.get_required_dict("MyDict")
entity = my_dict["MyData"]
if isinstance(entity, XRecord):
    user_record = UserRecord(entity)
    pprint(user_record.data)
```

Example 3: Store arbitrary binary data

```
my_dict = doc.rootdict.get_required_dict("MyDict")
xrecord = my_dict.add_xrecord("MyBinaryData")
with BinaryRecord(xrecord) as binary_record:
    # The content is stored as hex strings (e.g. ABBAFEFE...) in one or more
    # group code 310 tags.
    # A preceding group code 160 tag stores the data size in bytes.
    data = b"Store any binary data, even line breaks\r\n" * 20
    # compress data if required
    binary_record.data = zlib.compress(data, level=9)
```

Example 3: Get binary data back from the DICTIONARY object

```
entity = my_dict["MyBinaryData"]
if isinstance(entity, XRecord):
    binary_record = BinaryRecord(entity)
    print("\ncompressed data:")
    pprint(binary_record.data)

    print("\nuncompressed data:")
    pprint(zlib.decompress(binary_record.data))
```

Hint: Don't be fooled, the ability to save any binary data such as images, office documents, etc. in the DXF file doesn't impress AutoCAD, it simply ignores this data, this data only has a meaning for your application!

See also:

- [urecord module](#)
- [UserRecord class](#)

- *BinaryRecord* class

AppData

Application-Defined Data (AppData) was introduced in DXF R13/14 and is used by AutoCAD internally to store the handle to the *Extension Dictionary* and the *Reactors* in DXF entities. *Ezdx*f supports these kind of data storage for any AppID and the data is preserved by AutoCAD and BricsCAD, but I haven't found a way to access this data by AutoLISP or even the SDK. So I don't recommend this feature to store application defined data, because *Extended Data (XDATA)* and the *Extension Dictionary* are well documented and safe ways to attach custom data to entities.

```
# register your appid
APPID = "YOUR_UNIQUE_ID"
doc.appids.add(APPID)

# create a DXF entity
line = msp.add_line((0, 0), (1, 0))

# setting the data
line.set_app_data(APPID, [(300, "custom text"), (370, 4711), (460, 3.141592)])

# getting the data
if line.has_app_data(APPID):
    tags = line.get_app_data(APPID)
    print(f"{str(line)} has {len(tags)} tags of AppData for AppID {APPID!r}")
    for tag in tags:
        print(tag)
```

Printed output:

```
LINE(#30) has 3 tags of AppData for AppID 'YOUR_UNIQUE_ID'
(300, 'custom text')
(370, 4711)
(460, 3.141592)
```

6.5.27 Tutorial for MultiLeader

TODO ...

6.6 Howto

The Howto section show how to accomplish specific tasks with *ezdxf* in a straight forward way without teaching basics or internals, if you are looking for more information about the *ezdxf* internals look at the *Reference* section or if you want to learn how to use *ezdxf* go to the *Tutorials* section or to the *Basic Concepts* section.

6.6.1 General Document

General preconditions:

```
import sys
import ezdxf

try:
    doc = ezdxf.readfile("your_dxf_file.dxf")
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file.')
    sys.exit(2)
msp = doc.modelspace()
```

This works well with DXF files from trusted sources like AutoCAD or BricsCAD, for loading DXF files with minor or major flaws look at the [ezdxf.recover](#) module.

Load DXF Files with Structure Errors

If you know the files you will process have most likely minor or major flaws, use the [ezdxf.recover](#) module:

```
import sys
from ezdxf import recover

try: # low level structure repair:
    doc, auditor = recover.readfile(name)
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file: {name}.')
    sys.exit(2)

# DXF file can still have unrecoverable errors, but this is maybe
# just a problem when saving the recovered DXF file.
if auditor.has_errors:
    print(f'Found unrecoverable errors in DXF file: {name}.')
    auditor.print_error_report()
```

For more loading scenarios follow the link: [ezdxf.recover](#)

Set/Get Header Variables

ezdxf has an interface to get and set HEADER variables:

```
doc.header['VarName'] = value
value = doc.header['VarName']
```

See also:

[HeaderSection](#) and online documentation from Autodesk for available header variables.

Set DXF Drawing Units

The header variable \$INSUNITS defines the drawing units for the modelspace and therefore for the DXF document if no further settings are applied. The most common units are 6 for meters and 1 for inches.

Use this HEADER variables to setup the default units for CAD applications opening the DXF file. This setting is not relevant for `ezdxf` API calls, which are unitless for length values and coordinates and decimal degrees for angles (in most cases).

Sets drawing units:

```
doc.header['$INSUNITS'] = 6
```

For more information see section [DXF Units](#).

Create More Readable DXF Files (DXF Pretty Printer)

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

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

This creates a HTML file with a nicer layout than a plain text file, and handles are links between DXF entities, this simplifies the navigation between the DXF entities.

Changed in version 0.16: The `dxfpp` command was replaced by a sub-command of the `ezdxf` launcher.

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

positional arguments:
  FILE           DXF files pretty print

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

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

Calculate Extents for the Modelspace

Since `ezdxf` v0.16 exist a `ezdxf.bbox` module to calculate bounding boxes for DXF entities. This module makes the extents calculation very easy, but read the documentation for the `bbox` module to understand its limitations.

```
import ezdxf
from ezdxf import bbox

doc = ezdxf.readfile("your.dxf")
msp = doc.modelspace()

extents = bbox.extents(msp)
```

The returned `extents` is a `ezdxf.math.BoundingBox` object.

Set Initial View/Zoom for the Modelspace

To show an arbitrary location of the modelspace centered in the CAD application window, set the '`*Active' VPORT` to this location. The DXF attribute `dxf.center` defines the location in the modelspace, and the `dxf.height` specifies the area of the modelspace to view. Shortcut function:

```
doc.set_modelspace_vport(height=10, center=(10, 10))
```

New in version 0.16.

The new `ezdxf.zoom` module of `ezdxf` v0.16, makes this task much easier.

Setting the initial view to the extents of all entities in the modelspace:

```
import ezdxf
from ezdxf import zoom

doc = ezdxf.readfile("your.dxf")
msp = doc.modelspace()
zoom.extents(msp)
```

Setting the initial view to the extents of just some entities:

```
lines = msp.query("LINES")
zoom.objects(lines)
```

The `zoom` module also works for paperspace layouts.

Important: The `zoom` module uses the `bbox` module to calculate the bounding boxes for DXF entities. Read the documentation for the `bbox` module to understand its limitations and the bounding box calculation for large documents can take a while!

Add `ezdxf` Resources to Existing DXF Document

Add all `ezdxf` specific resources (line types, text- and dimension styles) to an existing DXF document:

```
import ezdxf
from ezdxf.tools.standards import setup_drawing

doc = ezdxf.readfile("your.dxf")
setup_drawing(doc, topics="all")
```

6.6.2 DXF Viewer

A360 Viewer Problems

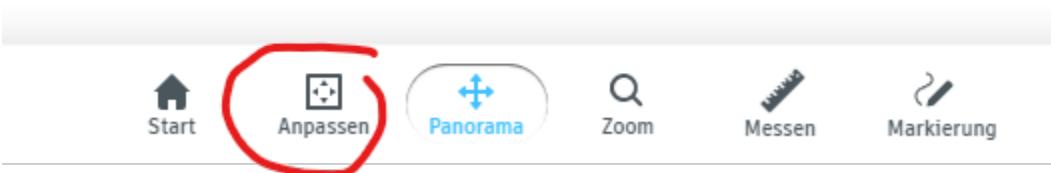
AutoDesk web service `A360` seems to be more picky than the AutoCAD desktop applications, may be it helps to use the latest DXF version supported by `ezdxf`, which is DXF R2018 (AC1032) in the year of writing this lines (2018).

DXF Entities Are Not Displayed in the Viewer

`ezdxf` does not automatically locate the main viewport of the modelspace at the entities, you have to perform the "Zoom to Extends" command, here in TrueView 2020:



And here in the Autodesk Online Viewer:



Add this line to your code to relocate the main viewport, adjust the *center* (in modelspace coordinates) and the *height* (in drawing units) arguments to your needs:

```
doc.set_modelspace_vport(height=10, center=(0, 0))
```

Show IMAGES/XREFS on Loading in AutoCAD

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

Thanks to David Booth:

If the filename in the IMAGEDEF contains the full path (absolute in AutoCAD) then it shows on loading, otherwise it won't display (reports as unreadable) until you manually reload using XREF manager.

A workaround (to show IMAGES on loading) appears to be to save the full file path in the DXF or save it as a DWG.

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

Set Initial View/Zoom for the Modelspace

See section "General Document": *Set Initial View/Zoom for the Modelspace*

6.6.3 DXF Content

General preconditions:

```
import sys
import ezdxf

try:
    doc = ezdxf.readfile("your_dxf_file.dxf")
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file.')
    sys.exit(2)
msp = doc.modelspace()
```

Get/Set Entity Color

The entity color is stored as *ACI* (AutoCAD Color Index):

```
aci = entity.dxf.color
```

Default value is 256 which means BYLAYER:

```
layer = doc.layers.get(entity.dxf.layer)
aci = layer.get_color()
```

The special `get_color()` method is required, because the color attribute `Layer.dxf.color` is misused as layer on/off flag, a negative color value means the layer is off.

ACI value 0 means BYBLOCK, which means the color from the block reference (INSERT entity).

Set color as ACI value as int in range [0, 256]:

```
entity.dxf.color = 1
```

The ACI value 7 has a special meaning, it is white on dark backgrounds and white on light backgrounds.

Get/Set Entity RGB Color

RGB true color values are supported since DXF R13 (AC1012), the 24-bit RGB value is stored as integer in the DXF attribute `true_color`:

```
# 24 bit binary value: 0bRRRRRRRRGGGGGGGGBBBBBBBB or hex value: 0xRRGGBB
# set true color value to red
entity.dxf.true_color = 0xFF0000
```

Use the helper functions from the `ezdxf.colors` module for RGB integer value handling:

```
from ezdxf import colors

entity.dxf.true_color = colors.rgb2int((0xFF, 0, 0))
r, g, b = colors.int2rgb(entity.dxf.true_color)
```

The RGB values of the AutoCAD default colors are not officially documented, but an accurate translation table is included in `ezdxf`:

```
# Warning: ACI value 256 (BYLAYER) raises an IndexError!
rgb24 = colors.DXF_DEFAULT_COLORS[aci]
print(f"RGB Hex Value: #{rgb24:06X}")
r, g, b = colors.int2rgb(rgb24)
print(f"RGB Channel Values: R={r:02X} G={g:02X} B={b:02X}")
```

If `color` and `true_color` values are set, BricsCAD and AutoCAD use the `true_color` value as display color for the entity.

Get/Set True Color as RGB-Tuple

Get/Set the true color value as (r, g, b)-tuple by the `rgb` property of the `DXFGraphic` entity:

```
# set true color value to red
entity.rgb = (0xFF, 0, 0)

# get true color values
r, g, b = entity.rgb
```

Get/Set Block Reference Attributes

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

Iterate over all appended attributes:

```
# get all INSERT entities with entity.dxf.name == "Part12"
blockrefs = msp.query('INSERT[name=="Part12"]')
if len(blockrefs):
    entity = blockrefs[0] # process first entity found
    for attrib in entity.attribs:
        if attrib.dxf.tag == "diameter": # identify attribute by tag
            attrib.dxf.text = "17mm" # change attribute content
```

Get attribute by tag:

```
diameter = entity.get_attrib('diameter')
if diameter is not None:
    diameter.dxf.text = "17mm"
```

Adding XDATA to Entities

Adding XDATA as list of tuples (group code, value) by `set_xdata()`, overwrites data if already present:

```
doc.appids.new('YOUR_APPID')  # IMPORTANT: create an APP ID entry

circle = msp.add_circle((10, 10), 100)
circle.set_xdata(
    'YOUR_APPID',
    [
        (1000, 'your_web_link.org'),
        (1002, '{'),
        (1000, 'some text'),
        (1002, '}'),
        (1071, 1),
        (1002, '}'),
        (1002, '}')
    ]
)
```

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

Method `get_xdata()` returns the extended data for an entity as `Tags` object.

See also:

Tutorial: [Storing Custom Data in DXF Files](#)

Get Overridden DIMSTYLE Values from DIMENSION

In general the `Dimension` styling and config attributes are stored in the `Dimstyle` entity, but every attribute can be overridden for each `DIMENSION` entity individually, get overwritten values by the `DimstyleOverride` object as shown in the following example:

```
for dimension in msp.query('DIMENSION'):
    dimstyle_override = dimension.override()  # requires v0.12
    dimtol = dimstyle_override['dimtol']
    if dimtol:
        print(f'{str(dimension)} has tolerance values:')
        dimtp = dimstyle_override['dimtp']
        dimtm = dimstyle_override['dimtm']
        print(f'Upper tolerance: {dimtp}')
        print(f'Lower tolerance: {dimtm}')
```

The `DimstyleOverride` object returns the value of the underlying `DIMSTYLE` objects if the value in `DIMENSION` was not overwritten, or `None` if the value was neither defined in `DIMSTYLE` nor in `DIMENSION`.

Override DIMSTYLE Values for DIMENSION

Same as above, the `DimstyleOverride` object supports also overriding `DIMSTYLE` values. But just overriding this values have no effect on the graphical representation of the `DIMENSION` entity, because CAD applications

just show the associated anonymous block which contains the graphical representation on the DIMENSION entity as simple DXF entities. Call the `render` method of the `DimstyleOverride` object to recreate this graphical representation by `ezdxf`, but `ezdxf` **does not** support all DIMENSION types and DIMVARS yet, and results **will differ** from AutoCAD or BricsCAD renderings.

```
dimstyle_override = dimension.override()
dimstyle_override.set_tolerance(0.1)

# delete associated geometry block
del doc.blocks[dimension.dxf.geometry]

# recreate geometry block
dimstyle_override.render()
```

6.7 FAQ

6.7.1 What is the Relationship between ezdxf, dxfwrite and dxfgabber?

In 2010 I started my first Python package for creating DXF documents called `dxfwrite`, this package can't read DXF files and writes only the DXF R12 (AC1009) version. While `dxfwrite` works fine, I wanted a more versatile package, that can read and write DXF files and maybe also supports newer DXF formats than DXF R12.

This was the start of the `ezdxf` package in 2011, but the progress was so slow, that I created a spin off in 2012 called `dxfgabber`, which implements only the reading part of `ezdxf`, which I needed for my work and I wasn't sure if `ezdxf` will ever be usable. Luckily in 2014 the first usable version of `ezdxf` could be released. The `ezdxf` package has all the features of `dxfwrite` and `dxfgabber` and much more, but with a different API. So `ezdxf` is not a drop-in replacement for `dxfgabber` or `dxfwrite`.

Since `ezdxf` can do all the things that `dxfwrite` and `dxfgabber` can do, I focused on the development of `ezdxf`, `dxfwrite` and `dxfgabber` are in maintenance mode only and will not get any new features, just bugfixes.

There are no advantages of `dxfwrite` over `ezdxf`, `dxfwrite` has the smaller memory footprint, but the `r12writer` add-on does the same job as `dxfwrite` without any in memory structures by writing direct to a stream or file and there is also no advantage of `dxfgabber` over `ezdxf` for normal DXF files the smaller memory footprint of `dxfgabber` is not noticeable and for really big files the `iterdxf` add-on does a better job.

6.7.2 Imported ezdxf package has no content. (readfile, new)

1. `AttributeError`: partially initialized module ‘ezdxf’ has no attribute ‘readfile’ (most likely due to a circular import)

Did you name your file/script “ezdxf.py”? This causes problems with circular imports. Renaming your file/script should solve this issue.

2. `AttributeError`: module ‘ezdxf’ has no attribute ‘readfile’

This could be a hidden permission error, for more information about this issue read Petr Zemeks article: <https://blog.petrzemek.net/2020/11/17/when-you-import-a-python-package-and-it-is-empty/>

6.7.3 How to add/edit ACIS based entities like 3DSOLID, REGION or SURFACE?

The BODY, 3DSOLID, SURFACE, REGION and so on, are stored as ACIS data embedded in the DXF file. The ACIS data is stored as SAT (text) format in the entity itself for DXF R2000-R2010 and as SAB (binary) format in the ACDSDATA section for DXF R2013+. `Ezdxf` can read SAT and SAB data, but only write SAT data.

The ACIS data is a proprietary format from [Spatial Inc.](#), and there exist no free available documentation or open source libraries to create or edit SAT or SAB data, and also *ezdxf* provides no functionality for creating or editing ACIS data.

The ACIS support provided by *ezdxf* is only useful for users have to have access to the ACIS SDK from [Spatial Inc.](#).

6.7.4 Are OLE/OLE2 entities supported?

TLDR; NO!

The Wikipedia definition of [OLE](#): Object Linking & Embedding (OLE) is a proprietary technology developed by Microsoft that allows embedding and linking to documents and other objects. For developers, it brought OLE Control Extension (OCX), a way to develop and use custom user interface elements. On a technical level, an OLE object is any object that implements the `IOleObject` interface, possibly along with a wide range of other interfaces, depending on the object's needs.

Therefore *ezdxf* does not support this entities in any way, this only work on Windows and with the required editing application installed. The binary data stored in the OLE objects cannot be used without the editing application.

In my opinion, using OLE objects in a CAD drawing is a very bad design decision that can and will cause problems opening these files in the future, even in AutoCAD on Windows when the required editing application is no longer available or the underlying technology is no longer supported.

All of this is unacceptable for a data storage format that should be accessed for many years or decades (e.g. construction drawings for buildings or bridges).

6.8 Reference

The [DXF Reference](#) is online available at Autodesk.

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

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

6.8.1 DXF Document

Document Management

Create New Drawings

`ezdxf.new(dxversion='AC1027', setup=False, units=6)` → Drawing

Create a new Drawing from scratch, `dxversion` can be either “AC1009” the official DXF version name or “R12” the AutoCAD release name.

`new()` can create drawings for following DXF versions:

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

The *units* argument defines the document and modelspace units. The header variable \$MEASUREMENT will be set according to the given *units*, 0 for inch, feet, miles, ... and 1 for metric units. For more information go to module `ezdxf.units`

Parameters

- **dxfversion** – DXF version specifier as string, default is “AC1027” respectively “R2013”
- **setup** – setup default styles, `False` for no setup, `True` to setup everything or a list of topics as strings, e.g. `["linetypes", "styles"]` to setup only some topics:

Topic	Description
linetypes	setup line types
styles	setup text styles
dimstyles	setup default <code>ezdxf</code> dimension styles
visualstyles	setup 25 standard visual styles

- **units** – document and modelspace units, default is 6 for meters

Open Drawings

Open DXF drawings from file system or text stream, byte stream usage is not supported.

DXF files prior to R2007 requires file encoding defined by header variable \$DWGCODEPAGE, DXF R2007 and later requires an UTF-8 encoding.

`ezdxf` supports reading of files for following DXF versions:

Version	Release	Encoding	Remarks
< AC1009		\$DWGCODEPAGE	pre AutoCAD R12 upgraded to AC1009
AC1009	R12	\$DWGCODEPAGE	AutoCAD R12
AC1012	R13	\$DWGCODEPAGE	AutoCAD R13 upgraded to AC1015
AC1014	R14	\$DWGCODEPAGE	AutoCAD R14 upgraded to AC1015
AC1015	R2000	\$DWGCODEPAGE	AutoCAD R2000
AC1018	R2004	\$DWGCODEPAGE	AutoCAD R2004
AC1021	R2007	UTF-8	AutoCAD R2007
AC1024	R2010	UTF-8	AutoCAD R2010
AC1027	R2013	UTF-8	AutoCAD R2013
AC1032	R2018	UTF-8	AutoCAD R2018

`ezdxf.readfile` (*filename*: str, *encoding*: str = `None`, *errors*: str = `"surrogateescape"`) → Drawing

Read the DXF document *filename* from the file-system.

This is the preferred method to load existing ASCII or Binary DXF files, the required text encoding will be detected automatically and decoding errors will be ignored.

Override encoding detection by setting argument *encoding* to the estimated encoding. (use Python encoding names like in the `open()` function).

If this function struggles to load the DXF document and raises a `DXFStructureError` exception, try the `ezdxf.recover.readfile()` function to load this corrupt DXF document.

Parameters

- **filename** – filename of the ASCII- or Binary DXF document
- **encoding** – use `None` for auto detect (default), or set a specific encoding like “`utf-8`”, argument is ignored for Binary DXF files
- **errors** – specify decoding error handler
 - “`surrogateescape`” to preserve possible binary data (default)
 - “`ignore`” to use the replacement char U+FFFD “`”` for invalid data
 - “`strict`” to raise an `UnicodeDecodeError` exception for invalid data

Raises

- `IOError` – not a DXF file or file does not exist
- `DXFStructureError` – for invalid or corrupted DXF structures
- `UnicodeDecodeError` – if *errors* is “`strict`” and a decoding error occurs

`ezdxf.read(stream: TextIO) → Drawing`

Read a DXF document from a text-stream. Open stream in text mode (`mode='rt'`) and set correct text encoding, the stream requires at least a `readline()` method.

Since DXF version R2007 (AC1021) file encoding is always “`utf-8`”, use the helper function `dxf_stream_info()` to detect the required text encoding for prior DXF versions. To preserve possible binary data in use `errors='surrogateescape'` as error handler for the import stream.

If this function struggles to load the DXF document and raises a `DXFStructureError` exception, try the `ezdxf.recover.read()` function to load this corrupt DXF document.

Parameters **stream** – input text stream opened with correct encoding

Raises `DXFStructureError` – for invalid or corrupted DXF structures

`ezdxf.readzip(zipfile: str, filename: str = None, errors: str="surrogateescape") → Drawing`

Load a DXF document specified by *filename* from a zip archive, or if *filename* is `None` the first DXF document in the zip archive.

Parameters

- **zipfile** – name of the zip archive
- **filename** – filename of DXF file, or `None` to load the first DXF document from the zip archive.
- **errors** – specify decoding error handler
 - “`surrogateescape`” to preserve possible binary data (default)
 - “`ignore`” to use the replacement char U+FFFD “`”` for invalid data
 - “`strict`” to raise an `UnicodeDecodeError` exception for invalid data

Raises

- `IOError` – not a DXF file or file does not exist or if `filename` is `None` - no DXF file found
- `DXFStructureError` – for invalid or corrupted DXF structures
- `UnicodeDecodeError` – if `errors` is “strict” and a decoding error occurs

`ezdxf.decode_base64(data: bytes, errors: str="surrogateescape") → Drawing`

Load a DXF document from base64 encoded binary data, like uploaded data to web applications.

Parameters

- `data` – DXF document base64 encoded binary data
- `errors` – specify decoding error handler
 - “surrogateescape” to preserve possible binary data (default)
 - “ignore” to use the replacement char U+FFFD “” for invalid data
 - “strict” to raise an `UnicodeDecodeError` exception for invalid data

Raises

- `DXFStructureError` – for invalid or corrupted DXF structures
- `UnicodeDecodeError` – if `errors` is “strict” and a decoding error occurs

Hint: This works well with DXF files from trusted sources like AutoCAD or BricsCAD, for loading DXF files with minor or major flaws look at the `ezdxf.recover` module.

Save Drawings

Save the DXF document to the file system by `Drawing` methods `save()` or `saveas()`. Write the DXF document to a text stream with `write()`, the text stream requires at least a `write()` method. Get required output encoding for text streams by property `Drawing.output_encoding`

Drawing Settings

The `HeaderSection` stores meta data like modelspace extensions, user name or saving time and current application settings, like actual layer, text style or dimension style settings. These settings are not necessary to process DXF data and therefore many of this settings are not maintained by `ezdxf` automatically.

Header variables set at new

\$ACADVER	DXF version
\$TDCREATE	date/time at creating the drawing
\$FINGERPRINTGUID	every drawing gets a GUID

Header variables updated at saving

\$TDUPDATE	actual date/time at saving
\$HANDSEED	next available handle as hex string
\$DWGCODEPAGE	encoding setting
\$VERSIONGUID	every saved version gets a new GUID

See also:

- Howto: *Set/Get Header Variables*
- Howto: *Set DXF Drawing Units*

Ezdx Metadata

New in version 0.17.

Store internal metadata like *ezdxf* version and creation time for a new created document as meta data in the DXF file. Only standard DXF features are used to store meta data and this meta data is preserved by Autodesk products, BricsCAD and of course *ezdxf*. Other 3rd party DXF libraries may remove this meta data.

For DXF R12 the meta data is stored as XDATA by AppID EZDXF in the model space BLOCK entity in the BLOCKS section.

For DXF R2000+ the meta data is stored in the “root” DICTIONARY in the OBJECTS section as a DICTIONARY object by the key EZDXF_META.

The MetaData object has a dict-like interface and can also store custom metadata:

```
metadata = doc.ezdx_metadata()

# set data
metadata["MY_CUSTOM_META_DATA"] = "a string with max. length of 254"

# get data, raises a KeyError() if key not exist
value = metadata["MY_CUSTOM_META_DATA"]

# get data, returns an empty string if key not exist
value = metadata.get("MY_CUSTOM_META_DATA")

# delete entry, raises a KeyError() if key not exist
del metadata["MY_CUSTOM_META_DATA"]

# discard entry, does not raise a KeyError() if key not exist
metadata.discard("MY_CUSTOM_META_DATA")
```

Keys and values are limited to strings with a max. length of 254 characters and line ending \n will be replaced by \P.

Keys used by *ezdxf*:

- WRITTEN_BY_EZDXF: *ezdxf* version and UTC time in ISO format
- CREATED_BY_EZDXF: *ezdxf* version and UTC time in ISO format

Example of the *ezdxf* marker string: 0.16.4b1 @ 2021-06-12T07:35:34.898808+00:00

```
class ezdx.document.MetaData
```

`MetaData.__contains__(key: str) → bool`
Returns `key` in self.

`MetaData.__getitem__(key: str) → str`
Returns the value for `self[key]`.

Raises `KeyError` – `key` does not exist

`MetaData.get(key: str, default: str = "") → str`
Returns the value for `key`. Returns `default` if `key` not exist.

`MetaData.__setitem__(key: str, value: str) → None`
Set `self[key]` to `value`.

`MetaData.__delitem__(key: str) → None`
Delete `self[key]`.

Raises `KeyError` – `key` does not exist

`MetaData.discard(key: str) → None`
Remove `key`, does **not** raise an exception if `key` not exist.

Drawing Object

`class ezdxf.document.Drawing`

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

`dxversion`

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

For supported DXF versions see *Document Management*

`acad_release`

The AutoCAD release name like 'R12' or 'R2000' for actual `dxversion`.

`encoding`

Text encoding of `Drawing`, the default encoding for new drawings is 'cp1252'. Starting with DXF R2007 (AC1021), DXF files are written as UTF-8 encoded text files, regardless of the attribute `encoding`. The text encoding can be changed to encodings listed below.

see also: *DXF File Encoding*

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

output_encoding

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

filename

Drawing filename, if loaded by `ezdxf.readfile()` else None.

rootdict

Reference to the root dictionary of the OBJECTS section.

header

Reference to the *HeaderSection*, get/set drawing settings as header variables.

entities

Reference to the EntitySection of the drawing, where all graphical entities are stored, but only from modelspace and the *active* paperspace layout. Just for your information: Entities of other paperspace layouts are stored as *BlockLayout* in the *BlocksSection*.

objects

Reference to the objects section, see also ObjectsSection.

blocks

Reference to the blocks section, see also BlocksSection.

tables

Reference to the tables section, see also TablesSection.

classes

Reference to the classes section, see also ClassesSection.

layouts

Reference to the layout manager, see also *Layouts*.

groups

Collection of all groups, see also *GroupCollection*.

requires DXF R13 or later

layers

Shortcut for Drawing.tables.layers

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

styles

Shortcut for Drawing.tables.styles

Reference to the styles table, see also Style.

dimstyles

Shortcut for Drawing.tables.dimstyles

Reference to the dimstyles table, see also DimStyle.

linetypes

Shortcut for Drawing.tables.linetypes

Reference to the linetypes table, see also Linetype.

views

Shortcut for Drawing.tables.views

Reference to the views table, see also View.

viewports

Shortcut for Drawing.tables.viewports

Reference to the viewports table, see also [Viewport](#).

ucs

Shortcut for `Drawing.tables.ucs`

Reference to the ucs table, see also [UCS](#).

appids

Shortcut for `Drawing.tables.appids`

Reference to the appids table, see also [AppID](#).

materials

MaterialCollection of all Material objects.

mline_styles

MLineStyleCollection of all `MLinestyle` objects.

mleader_styles

MLeaderStyleCollection of all `MLeaderStyle` objects.

units

Get and set the document/modelspace base units as enum, for more information read this: [DXF Units](#).

save (*encoding: str = None, fmt: str = 'asc'*) → None

Write drawing to file-system by using the `filename` attribute as filename. Override file encoding by argument `encoding`, handle with care, but this option allows you to create DXF files for applications that handles file encoding different than AutoCAD.

Parameters

- **encoding** – override default encoding as Python encoding string like 'utf-8'
- **fmt** – 'asc' for ASCII DXF (default) or 'bin' for Binary DXF

saveas (*filename: Union[str, Path], encoding: str = None, fmt: str = 'asc'*) → None

Set `Drawing` attribute `filename` to `filename` and write drawing to the file system. Override file encoding by argument `encoding`, handle with care, but this option allows you to create DXF files for applications that handles file encoding different than AutoCAD.

Parameters

- **filename** – file name as string
- **encoding** – override default encoding as Python encoding string like 'utf-8'
- **fmt** – 'asc' for ASCII DXF (default) or 'bin' for Binary DXF

write (*stream: Union[TextIO, BinaryIO], fmt: str = 'asc'*) → None

Write drawing as ASCII DXF to a text stream or as Binary DXF to a binary stream. For DXF R2004 (AC1018) and prior open stream with drawing `encoding` and `mode='wt'`. For DXF R2007 (AC1021) and later use `encoding='utf-8'`, or better use the later added `Drawing` property `output_encoding` which returns the correct encoding automatically. The correct and required error handler is `errors='dxffreplace'`!

If writing to a `StringIO` stream, use `Drawing.encode()` to encode the result string from `StringIO.getvalue()`:

```
binary = doc.encode(stream.getvalue())
```

Parameters

- **stream** – output text stream or binary stream

- **fmt** – 'asc' for ASCII DXF (default) or 'bin' for binary DXF

encode_base64 () → bytes

Returns DXF document as base64 encoded binary data.

encode (s: str) → bytes

Encode string *s* with correct encoding and error handler.

query (query: str = '*') → ezdxf.query.EntityQuery

Entity query over all layouts and blocks, excluding the OBJECTS section.

Parameters **query** – query string

See also:

Entity Query String and *Retrieve entities by query language*

groupby (dxftattrib=”, key=None) → dict

Groups DXF entities of all layouts and blocks (excluding the OBJECTS section) by a DXF attribute or a key function.

Parameters

- **dxftattrib** – grouping DXF attribute like 'layer'
- **key** – key function, which accepts a DXFEntity as argument and returns a hashable grouping key or None to ignore this entity.

See also:

groupby () documentation

modelspace () → ezdxf.layouts.layout.Modelspace

Returns the modelspace layout, displayed as 'Model' tab in CAD applications, defined by block record named 'Model_Space'.

layout (name: str = None) → Layout

Returns paperspace layout *name* or returns first layout in tab order if *name* is None.

active_layout () → Layout

Returns the active paperspace layout, defined by block record name 'Paper_Space'.

layout_names () → Iterable[str]

Returns all layout names (modelspace 'Model' included) in arbitrary order.

layout_names_in_taborder () → Iterable[str]

Returns all layout names in tab order, layout "Model" (model space) is always the first name.

new_layout (name, dxftattribs=None) → Layout

Create a new paperspace layout *name*. Returns a *Layout* object. DXF R12 (AC1009) supports only one paperspace layout, only the active paperspace layout is saved, other layouts are dismissed.

Parameters

- **name** – unique layout name
- **dxftattribs** – additional DXF attributes for the DXFLayout entity

Raises DXFValueError – *Layout* *name* already exist

delete_layout (name: str) → None

Delete paper space layout *name* and all entities owned by this layout. Available only for DXF R2000 or later, DXF R12 supports only one paperspace and it can't be deleted.

add_image_def(filename: str, size_in_pixel: Tuple[int, int], name=None)

Add an image definition to the objects section.

Add an `ImageDef` entity to the drawing (objects section). `filename` is the image file name as relative or absolute path and `size_in_pixel` is the image size in pixel as (x, y) tuple. To avoid dependencies to external packages, `ezdxf` can not determine the image size by itself. Returns a `ImageDef` entity which is needed to create an image reference. `name` is the internal image name, if set to `None`, name is auto-generated.

Absolute image paths works best for AutoCAD but not really good, you have to update external references manually in AutoCAD, which is not possible in TrueView. If the drawing units differ from 1 meter, you also have to use: `set_raster_variables()`.

Parameters

- `filename` – image file name (absolute path works best for AutoCAD)
- `size_in_pixel` – image size in pixel as (x, y) tuple
- `name` – image name for internal use, `None` for using `filename` as name (best for AutoCAD)

See also:

Tutorial for Image and ImageDef

set_raster_variables(frame: int = 0, quality: int = 1, units: str = 'm')

Set raster variables.

Parameters

- `frame` – 0 = do not show image frame; 1 = show image frame
- `quality` – 0 = draft; 1 = high
- `units` – units for inserting images. This defines the real world unit for one drawing unit for the purpose of inserting and scaling images with an associated resolution.

mm	Millimeter
cm	Centimeter
m	Meter (ezdxf default)
km	Kilometer
in	Inch
ft	Foot
yd	Yard
mi	Mile

set_wipeout_variables(frame=0)

Set wipeout variables.

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

add_underlay_def(filename: str, fmt: str = 'ext', name: str = None)

Add an `UnderlayDef` entity to the drawing (OBJECTS section). `filename` is the underlay file name as relative or absolute path and `fmt` as string (pdf, dwf, dgn). The underlay definition is required to create an underlay reference.

Parameters

- `filename` – underlay file name
- `fmt` – file format as string '`pdf`' | '`dwf`' | '`dgn`' or '`ext`' for getting file format from filename extension
- `name` – pdf format = page number to display; dgn format = '`default`'; dwf: ????

See also:

Tutorial for Underlay and UnderlayDefinition

add_xref_def (*filename*: str, *name*: str, *flags*: int = 20)

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

Parameters

- **filename** – external reference filename
- **name** – name of the xref block
- **flags** – block flags

layouts_and_blocks () → Iterator[GenericLayoutType]

Iterate over all layouts (modelspace and paperspace) and all block definitions.

chain_layouts_and_blocks () → Iterator[DXFEntity]

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

reset_fingerprint_guid()

Reset fingerprint GUID.

reset_version_guid()

Reset version GUID.

set_modelspace_vport (*height*, *center*=(0, 0), *, *dxfattribs*=None) → VPort

Set initial view/zoom location for the modelspace, this replaces the current “*Active” viewport configuration ([VPort](#)).

Parameters

- **height** – modelspace area to view
- **center** – modelspace location to view in the center of the CAD application window.
- **dxfattribs** – additional DXF attributes for the VPORT entity

Changed in version 0.17.2: added argument *dxfattribs* to pass additional DXF attributes to the VPORT entity

audit () → ezdxf.audit.Auditor

Checks document integrity and fixes all fixable problems, not fixable problems are stored in `Auditor.errors`.

If you are messing around with internal structures, call this method before saving to be sure to export valid DXF documents, but be aware this is a long running task.

validate (*print_report*=True) → bool

Simple way to run an audit process. Fixes all fixable problems, return `False` if not fixable errors occurs, to get more information about not fixable errors use `audit()` method instead.

Parameters `print_report` – print report to stdout

Returns: `True` if no errors occurred

ezdxf_metadata () → MetaData

Returns the `ezdxf.ezdxf.Document.MetaData` object, which manages `ezdxf` and custom metadata in DXF files. For more information see: [Ezdx Metadata](#).

New in version 0.17.

Recover

New in version v0.14.

This module provides functions to “recover” ASCII DXF documents with structural flaws, which prevents the regular `ezdxf.read()` and `ezdxf.readfile()` functions to load the document.

The `read()` and `readfile()` functions will repair as much flaws as possible and run the required audit process automatically afterwards and return the result of this audit process:

```
import sys
import eздxf
from eздxf import recover

try:
    doc, auditor = recover.readfile("messy.dxf")
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except eздxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file.')
    sys.exit(2)

# DXF file can still have unrecoverable errors, but this is maybe just
# a problem when saving the recovered DXF file.
if auditor.has_errors:
    auditor.print_error_report()
```

This efforts cost some time, loading the DXF document with `ezdxf.read()` or `ezdxf.readfile()` will be faster.

Warning: This module will load DXF files which have decoding errors, most likely binary data stored in XRECORD entities, these errors are logged as unrecoverable `AuditError.DECODE_ERRORS` in the `Auditor.errors` attribute, but no `DXFStructureError` exception will be raised, because for many use cases this errors can be ignored.

Writing such files back with `ezdxf` may create **invalid** DXF files, or at least some **information will be lost** - handle with care!

To avoid this problem use `recover.readfile(filename, errors='strict')` which raises an `UnicodeDecodeError` exception for such binary data. Catch the exception and handle this DXF files as unrecoverable.

Loading Scenarios

1. It will work

Mostly DXF files from AutoCAD or BricsCAD (e.g. for In-house solutions):

```
try:
    doc = eздxf.readfile(name)
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except e Zukxf.DXFStructureError:
```

(continues on next page)

(continued from previous page)

```
print(f'Invalid or corrupted DXF file: {name}.')
sys.exit(2)
```

2. DXF file with minor flaws

DXF files have only minor flaws, like undefined resources:

```
try:
    doc = eздxf.readfile(name)
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except eздxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file: {name}.')
    sys.exit(2)

auditor = doc.audit()
if auditor.has_errors:
    auditor.print_error_report()
```

3. Try Hard

From trusted and untrusted sources but with good hopes, the worst case works like a cache miss, you pay for the first try and pay the extra fee for the recover mode:

```
try: # Fast path:
    doc = eздxf.readfile(name)
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
# Catch all DXF errors:
except eздxf.DXFError:
    try: # Slow path including fixing low level structures:
        doc, auditor = recover.readfile(name)
    except eздxf.DXFStructureError:
        print(f'Invalid or corrupted DXF file: {name}.')
        sys.exit(2)

# DXF file can still have unrecoverable errors, but this is maybe
# just a problem when saving the recovered DXF file.
if auditor.has_errors:
    print(f'Found unrecoverable errors in DXF file: {name}.')
    auditor.print_error_report()
```

4. Just use the slow recover module

Untrusted sources and expecting many invalid or corrupted DXF files, you always pay an extra fee for the recover mode:

```
try: # Slow path including fixing low level structures:
    doc, auditor = recover.readfile(name)
```

(continues on next page)

(continued from previous page)

```

except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file: {name}.')
    sys.exit(2)

# DXF file can still have unrecoverable errors, but this is maybe
# just a problem when saving the recovered DXF file.
if auditor.has_errors:
    print(f'Found unrecoverable errors in DXF file: {name}.')
    auditor.print_error_report()

```

5. Unrecoverable Decoding Errors

If files contain binary data which can not be decoded by the document encoding, it is maybe the best to ignore this files, this works in normal and recover mode:

```

try:
    doc, auditor = recover.readfile(name, errors='strict')
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file: {name}.')
    sys.exit(2)
except UnicodeDecodeError:
    print(f'Decoding error in DXF file: {name}.')
    sys.exit(3)

```

6. Ignore/Locate Decoding Errors

Sometimes ignoring decoding errors can recover DXF files or at least you can detect where the decoding errors occur:

```

try:
    doc, auditor = recover.readfile(name, errors='ignore')
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file: {name}.')
    sys.exit(2)
if auditor.has_errors:
    auditor.print_report()

```

The error messages with code `AuditError.DECODING_ERROR` shows the approximate line number of the decoding error: “Fixed unicode decoding error near line: xxx.”

Hint: This functions can handle only ASCII DXF files!

`ezdxf.recover.readfile(filename: Union[str, pathlib.Path], errors: str = 'surrogateescape') → Tuple[Drawing, Auditor]`

Read a DXF document from file system similar to `ezdxf.readfile()`, but this function will repair as much flaws as possible, runs the required audit process automatically the DXF document and the Auditor.

Parameters

- **filename** – file-system name of the DXF document to load
- **errors** – specify decoding error handler
 - “surrogateescape” to preserve possible binary data (default)
 - “ignore” to use the replacement char U+FFFD “” for invalid data
 - “strict” to raise an `UnicodeDecodeError` exception for invalid data

Raises

- `DXFStructureError` – for invalid or corrupted DXF structures
- `UnicodeDecodeError` – if `errors` is “strict” and a decoding error occurs

`ezdxf.recover.read(stream: BinaryIO, errors: str = 'surrogateescape') → Tuple[Drawing, Auditor]`

Read a DXF document from a binary-stream similar to `ezdxf.read()`, but this function will detect the text encoding automatically and repair as much flaws as possible, runs the required audit process afterwards and returns the DXF document and the Auditor.

Parameters

- **stream** – data stream to load in binary read mode
- **errors** – specify decoding error handler
 - “surrogateescape” to preserve possible binary data (default)
 - “ignore” to use the replacement char U+FFFD “” for invalid data
 - “strict” to raise an `UnicodeDecodeError` exception for invalid data

Raises

- `DXFStructureError` – for invalid or corrupted DXF structures
- `UnicodeDecodeError` – if `errors` is “strict” and a decoding error occurs

`ezdxf.recover.explore(filename: Union[str, pathlib.Path], errors: str = 'ignore') → Tuple[Drawing, Auditor]`

Read a DXF document from file system similar to `readfile()`, but this function will use a special tag loader, which synchronise the tag stream if invalid tags occur. This function is intended to load corrupted DXF files and should only be used to explore such files, data loss is very likely.

Parameters

- **filename** – file-system name of the DXF document to load
- **errors** – specify decoding error handler
 - “surrogateescape” to preserve possible binary data (default)
 - “ignore” to use the replacement char U+FFFD “” for invalid data
 - “strict” to raise an `UnicodeDecodeError` exception for invalid data

Raises

- `DXFStructureError` – for invalid or corrupted DXF structures
- `UnicodeDecodeError` – if `errors` is “strict” and a decoding error occurs

6.8.2 DXF Structures

Sections

Header Section

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

See also:

DXF Internals: *HEADER Section*

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

`custom_vars`

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

```
__len__() → int
```

Returns count of header variables.

```
__contains__(key) → bool
```

Returns True if header variable `key` exist.

```
varnames() → KeysView[KT]
```

Returns an iterable of all header variable names.

```
get(key: str, default: Any = None) → Any
```

Returns value of header variable `key` if exist, else the `default` value.

```
__getitem__(key: str) → Any
```

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

```
__setitem__(key: str, value: Any) → None
```

Set header variable `key` to `value` by index operator like: `drawing.header['$ANGDIR'] = 1`

```
__delitem__(key: str) → None
```

Delete header variable `key` by index operator like: `del drawing.header['$ANGDIR']`

```
class ezdxf.sections.header.CustomVars
```

Stores custom properties in the DXF header as `$CUSTOMPROPERTYTAG` and `$CUSTOMPROPERTY` values. Custom properties are just supported by DXF R2004 (AC1018) or later. `ezdxf` can create custom properties at older DXF versions, but AutoCAD will not show this properties.

`properties`

List of custom drawing properties, stored as string tuples (`tag, value`). Multiple occurrence of the same custom tag is allowed, but not well supported by the interface. This is a standard python list and it is save to change this list as long you store just tuples of strings in the format (`tag, value`).

```
__len__() → int
```

Count of custom properties.

```
__iter__() → Iterable[Tuple[str, str]]
```

Iterate over all custom properties as (`tag, value`) tuples.

```
clear() → None
```

Remove all custom properties.

```
get(tag: str, default: str = None)
```

Returns the value of the first custom property `tag`.

has_tag (*tag: str*) → bool

Returns True if custom property *tag* exist.

append (*tag: str, value: str*) → None

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

replace (*tag: str, value: str*) → None

Replaces the value of the first custom property *tag* by a new *value*.

Raises DXFValueError if *tag* does not exist.

remove (*tag: str, all: bool = False*) → None

Removes the first occurrence of custom property *tag*, removes all occurrences if *all* is True.

Raises :class:`DXFValueError` if *tag* does not exist.

Classes Section

The CLASSES section in DXF files holds the information for application-defined classes whose instances appear in *Layout* objects. As usual package user there is no need to bother about CLASSES.

See also:

DXF Internals: *CLASSES Section*

class ezdxf.sections.classes.**ClassesSection****classes**

Storage of all *DXFClass* objects, they are not stored in the entities database, because CLASS has no handle attribute.

register (*classes: Iterable[DXFClass]*)**add_class** (*name: str*)

Register a known class by *name*.

get (*name: str*) → *DXFClass*

Returns the first class matching *name*.

Storage key is the (*name, cpp_class_name*) tuple, because there are some classes with the same name but different *cpp_class_names*.

add_required_classes (*name: str*) → *DXFClass*

Add all required CLASS definitions for *dxfversion*.

update_instance_counters () → None

Update CLASS instance counter for all registered classes, requires DXF R2004+.

class ezdxf.entities.**DXFClass**

Information about application-defined classes.

dxfl.name

Class DXF record name.

dxfl.cpp_class_name

C++ class name. Used to bind with software that defines object class behavior.

dxfl.app_name

Application name. Posted in Alert box when a class definition listed in this section is not currently loaded.

dx.f.flags

Proxy capabilities flag

0	No operations allowed (0)
1	Erase allowed (0x1)
2	Transform allowed (0x2)
4	Color change allowed (0x4)
8	Layer change allowed (0x8)
16	Linetype change allowed (0x10)
32	Linetype scale change allowed (0x20)
64	Visibility change allowed (0x40)
128	Cloning allowed (0x80)
256	Lineweight change allowed (0x100)
512	Plot Style Name change allowed (0x200)
895	All operations except cloning allowed (0x37F)
1023	All operations allowed (0x3FF)
1024	Disables proxy warning dialog (0x400)
32768	R13 format proxy (0x8000)

dx.f.instance_count

Instance count for a custom class.

dx.f.was_a_proxy

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

dx.f.is_an_entity

Set to 1 if class was derived from the *DXFGraphic* class and can reside in layouts. If 0, instances may appear only in the OBJECTS section.

key

Unique name as (name, cpp_class_name) tuple.

Tables Section

The TABLES section is the home of all TABLE objects of a DXF document.

See also:

DXF Internals: *TABLES Section*

class ezdxf.sections.tables.TablesSection
layers

LayerTable object for *Layer* objects

linetypes

LineTypesTable object for *Linetype* objects

styles

TextstyleTable object for *Textstyle* objects

dimstyles

DimStyleTable object for *DimStyle* objects

appids

AppIDTable object for *AppID* objects

ucs

UCSTable object for *UCSTable* objects

views

ViewTable object for *View* objects

viewports

ViewportTable object for *VPort* objects

block_records

BlockRecordTable object for *BlockRecord* objects

Blocks Section

The BLOCKS section is the home all block definitions (*BlockLayout*) of a DXF document.

See also:

DXF Internals: *BLOCKS Section* and *Block Management Structures*

class ezdxf.sections.blocks.**BlocksSection**

__iter__ () → Iterable[*BlockLayout*]
Iterable of all *BlockLayout* objects.

__contains__ (name: str) → bool
Returns True if *BlockLayout* name exist.

__getitem__ (name: str) → *BlockLayout*
Returns *BlockLayout* name, raises DXFKeyError if name not exist.

__delitem__ (name: str) → None
Deletes *BlockLayout* name and all of its content, raises DXFKeyError if name not exist.

get (self, name: str, default=None) → *BlockLayout*
Returns *BlockLayout* name, returns default if name not exist.

new (name: str, base_point: Vertex = (0, 0, 0), dxfattribs: dict = None) → *BlockLayout*
Create and add a new *BlockLayout*, name is the BLOCK name, base_point is the insertion point of the BLOCK.

new_anonymous_block (type_char: str = "U", base_point: Vertex = (0, 0, 0)) → *BlockLayout*
Create and add a new anonymous *BlockLayout*, type_char is the BLOCK type, base_point is the insertion point of the BLOCK.

type_char	Anonymous Block Type
'U'	'*U###' anonymous BLOCK
'E'	'*E###' anonymous non-uniformly scaled BLOCK
'X'	'*X###' anonymous HATCH graphic
'D'	'*D###' anonymous DIMENSION graphic
'A'	'*A###' anonymous GROUP
'T'	'*T###' anonymous block for ACAD_TABLE content

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

delete_block (name: str, safe: bool = True) → None
Delete block. If save is True, check if block is still referenced.

Parameters

- **name** – block name (case insensitive)
- **safe** – check if block is still referenced or special block without explicit references

Raises

- `DXFKeyError` – if block not exists
- `DXFBlockInUseError` – if block is still referenced, and save is True

`delete_all_blocks()`

Delete all blocks without references except modelspace- or paperspace layout blocks, special arrow- and anonymous blocks (DIMENSION, ACAD_TABLE).

Warning: There could exist undiscovered references to blocks which are not documented in the DXF reference, hidden in extended data sections or application defined data, which could produce invalid DXF documents if such referenced blocks will be deleted.

Changed in version 0.14: removed unsafe mode

Entities Section

The ENTITIES section is the home of all *Modelspace* and active *Paperspace* layout entities. This is a real section in the DXF file, for *ezdxf* the *EntitySection* is just a proxy for modelspace and the active paperspace linked together.

See also:

DXF Internals: *ENTITIES Section*

`class ezdxf.sections.entities.EntitySection`

- `__iter__()` → Iterable[DXFEntity]
Iterable for all entities of modelspace and active paperspace.
- `__len__()` → int
Returns count of all entities of modelspace and active paperspace.

Objects Section

The OBJECTS section is the home of all none graphical objects of a DXF document. The OBJECTS section is accessible by `Drawing.objects`.

Convenience methods of the `Drawing` object to create required structures in the OBJECTS section:

- IMAGEDEF: `add_image_def()`
- UNDERLAYDEF: `add_underlayer_def()`
- RASTERVARIABLES: `set_raster_variables()`
- WIPEOUTVARIABLES: `set_wipeout_variables()`

See also:

DXF Internals: *OBJECTS Section*

```
class ezdxf.sections.objects.ObjectsSection
```

rootdict

the named DICTIONARY.

Type Returns the root DICTIONARY, or as AutoCAD calls it

__len__ () → int

Returns count of DXF objects.

__iter__ ()

Returns iterable of all DXF objects in the OBJECTS section.

__getitem__ (index) → DXFObject

Get entity at *index*.

The underlying data structure for storing DXF objects is organized like a standard Python list, therefore *index* can be any valid list indexing or slicing term, like a single index `objects[-1]` to get the last entity, or an index slice `objects[:10]` to get the first 10 or less objects as List [DXFObject].

__contains__ (entity)

Returns True if *entity* stored in OBJECTS section.

Parameters **entity** – DXFObject or handle as hex string

query (query: str = '*') → ezdxf.query.EntityQuery

Get all DXF objects matching the *Entity Query String*.

add_dictionary (owner: str = '0', hard_owned: bool = True) → ezdxf.entities.dictionary.Dictionary

Add new *Dictionary* object.

Parameters

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

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

Add new *DictionaryWithDefault* object.

Parameters

- **owner** – handle to owner as hex string.
- **default** – handle to default entry.
- **hard_owned** – True to treat entries as hard owned.

add_dictionary_var (owner: str = '0', value: str = '') → DictionaryVar

Add a new *DictionaryVar* object.

Parameters

- **owner** – handle to owner as hex string.
- **value** – value as string

add_geodata (owner: str = '0', dxfattribs: dict = None) → GeoData

Creates a new GeoData entity and replaces existing ones. The GEODATA entity resides in the OBJECTS section and NOT in the layout entity space and it is linked to the layout by an extension dictionary located in BLOCK_RECORD of the layout.

The GEODATA entity requires DXF version R2010+. The DXF Reference does not document if other layouts than model space supports geo referencing, so getting/setting geo data may only make sense for the model space layout, but it is also available in paper space layouts.

Parameters

- **owner** – handle to owner as hex string
- **dxfattribs** – DXF attributes for `GeoData` entity

add_image_def (*filename*: str, *size_in_pixel*: Tuple[int, int], *name*=None) → ImageDef

Add an image definition to the objects section.

Add an `ImageDef` entity to the drawing (objects section). *filename* is the image file name as relative or absolute path and *size_in_pixel* is the image size in pixel as (x, y) tuple. To avoid dependencies to external packages, `ezdxf` can not determine the image size by itself. Returns a `ImageDef` entity which is needed to create an image reference. *name* is the internal image name, if set to None, name is auto-generated.

Absolute image paths works best for AutoCAD but not really good, you have to update external references manually in AutoCAD, which is not possible in TrueView. If the drawing units differ from 1 meter, you also have to use: `set_raster_variables()`.

Parameters

- **filename** – image file name (absolute path works best for AutoCAD)
- **size_in_pixel** – image size in pixel as (x, y) tuple
- **name** – image name for internal use, None for using filename as name (best for AutoCAD)

add_placeholder (*owner*: str = '0') → Placeholder

Add a new `Placeholder` object.

Parameters **owner** – handle to owner as hex string.

add_underlay_def (*filename*: str, *fmt*: str = 'pdf', *name*: str = None) → UnderlayDefinition

Add an `UnderlayDefinition` entity to the drawing (OBJECTS section). *filename* is the underlay file name as relative or absolute path and *fmt* as string (pdf, dwf, dgn). The underlay definition is required to create an underlay reference.

Parameters

- **filename** – underlay file name
- **fmt** – file format as string 'pdf' | 'dwf' | 'dgn'
- **name** – pdf format = page number to display; dgn format = 'default'; dwf: ????

add_xrecord (*owner*: str = '0') → XRecord

Add a new `XRecord` object.

Parameters **owner** – handle to owner as hex string.

set_raster_variables (*frame*: int = 0, *quality*: int = 1, *units*: str = 'm') → None

Set raster variables.

Parameters

- **frame** – 0 = do not show image frame; 1 = show image frame
- **quality** – 0 = draft; 1 = high
- **units** – units for inserting images. This defines the real world unit for one drawing unit for the purpose of inserting and scaling images with an associated resolution.

mm	Millimeter
cm	Centimeter
m	Meter (ezdxf default)
km	Kilometer
in	Inch
ft	Foot
yd	Yard
mi	Mile

(internal API), public interface `set_raster_variables()`

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

Set wipeout variables.

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

(internal API)

Tables

Table Classes

Generic Table Class

class `ezdxf.sections.table.Table`

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

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

Unified table entry key.

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

Returns True if an table entry `name` exist.

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

Returns True if an table entry `name` exist.

__len__ () → int

Count of table entries.

__iter__ () → Iterable[DXFEntity]

Iterable of all table entries.

new (`name: str, dxfattribs: dict = None`) → DXFEntity

Create a new table entry `name`.

Parameters

- **name** – name of table entry, case insensitive
- **dxfattribs** – additional DXF attributes for table entry

get (`name: str`) → DXFEntity

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

remove (`name: str`) → None

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

duplicate_entry (`name: str, new_name: str`) → DXFEntity

Returns a new table entry `new_name` as copy of `name`, replaces entry `new_name` if already exist.

Raises `DXFValueError` – `name` does not exist

Layer Table

class `ezdxf.sections.table.LayerTable`

Subclass of `Table`.

Collection of `Layer` objects.

add (`name: str`, *, `color: int = 256`, `true_color: int = None`, `linetype: str = 'Continuous'`, `lineweight: int = -1`, `plot: bool = True`, `transparency: Optional[float] = None`, `dxfattribs: Dict[KT, VT] = None`) → `Layer`
Add a new `Layer`.

Parameters

- `name (str)` – layer name
- `color (int)` – *AutoCAD Color Index (ACI)* value, default is BYLAYER
- `true_color (int)` – true color value, use `ezdxf.rgb2int()` to create `int` values from RGB values
- `linetype (str)` – line type name, default is “Continuous”
- `lineweight (int)` – line weight, default is BYLAYER
- `plot (bool)` – plot layer as bool, default is True
- `transparency` – transparency value in the range [0, 1], where 1 is 100% transparent and 0 is opaque
- `dxfattribs (dict)` – additional DXF attributes

New in version 0.17.

Linetype Table

class `ezdxf.sections.table.LinetypeTable`

Subclass of `Table`.

Collection of `Linetype` objects.

add (`name: str`, `pattern: Union[List[float], str]`, *, `description: str = ''`, `length: float = 0.0`, `dxfattribs: Dict[KT, VT] = None`) → `Linetype`
Add a new line type entry. The simple line type pattern is a list of floats [`total_pattern_length`, `elem1`, `elem2`, ...] where an element > 0 is a line, an element < 0 is a gap and an element == 0.0 is a dot. The definition for complex line types are strings, like: 'A, .5, -.2, ["GAS", STANDARD, S=.1, U=0.0, X=-0.1, Y=-.05], -.25' similar to the line type definitions stored in the line definition `.lin` files, for more information see the tutorial about complex line types. Be aware that not many CAD applications and DXF viewers support complex linetypes.

See also:

- Tutorial for simple line types
- Tutorial for complex line types

Parameters

- `name (str)` – line type name

- **pattern** – line type pattern as list of floats or as a string
- **description** (*str*) – line type description, optional
- **length** (*float*) – total pattern length, only for complex line types required
- **dxfattribs** (*dict*) – additional DXF attributes

New in version 0.17.

Style Table

class `ezdxf.sections.table.TextstyleTable`

Subclass of `Table`.

Collection of `Textstyle` objects.

add (*name: str*, *, *font: str*, *dxfattribs: Dict[KT, VT] = None*) → `Textstyle`

Add a new text style entry for TTF fonts. The entry must not yet exist, otherwise an `DXFTableEntryError` exception will be raised.

Finding the TTF font files is the task of the DXF viewer and each viewer is different (hint: support files).

Parameters

- **name** (*str*) – text style name
- **font** (*str*) – TTF font file name like “Arial.ttf”, the real font file name from the file system is required and remember only Windows is case insensitive.
- **dxfattribs** (*dict*) – additional DXF attributes

New in version 0.17.

add_shx (*shx_file: str*, *, *dxfattribs: Dict[KT, VT] = None*) → `Textstyle`

Add a new shape font (SHX file) entry. These are special text style entries and have no name. The entry must not yet exist, otherwise an `DXFTableEntryError` exception will be raised.

Finding the SHX files is the task of the DXF viewer and each viewer is different (hint: support files).

Parameters

- **shx_file** (*str*) – shape file name like “gdt.shx”
- **dxfattribs** (*dict*) – additional DXF attributes

New in version 0.17.

get_shx (*shx_file: str*) → `Textstyle`

Get existing entry for a shape file (SHX file), or create a new entry.

Finding the SHX files is the task of the DXF viewer and each viewer is different (hint: support files).

Parameters **shx_file** (*str*) – shape file name like “gdt.shx”

find_shx (*shx_file: str*) → `Optional[Textstyle]`

Find the shape file (SHX file) text style table entry, by a case insensitive search.

A shape file table entry has no name, so you have to search by the font attribute.

Parameters **shx_file** (*str*) – shape file name like “gdt.shx”

DimStyle Table

```
class ezdxf.sections.table.DimStyleTable
    Subclass of Table.
```

Collection of [DimStyle](#) objects.

add (*name: str*, *, *dxfattribs: Dict[KT, VT] = None*) → DimStyle
Add a new dimension style table entry.

Parameters

- **name** (*str*) – dimension style name
- **dxfattribs** (*dict*) – DXF attributes

New in version 0.17.

AppID Table

```
class ezdxf.sections.table.AppIDTable
    Subclass of Table.
```

Collection of [AppID](#) objects.

add (*name: str*, *, *dxfattribs: Dict[KT, VT] = None*) → AppID
Add a new appid table entry.

Parameters

- **name** (*str*) – appid name
- **dxfattribs** (*dict*) – DXF attributes

New in version 0.17.

UCS Table

```
class ezdxf.sections.table.UCSTable
    Subclass of Table.
```

Collection of [UCSTableEntry](#) objects.

add (*name: str*, *, *dxfattribs: Dict[KT, VT] = None*) → UCSTableEntry
Add a new UCS table entry.

Parameters

- **name** (*str*) – UCS name
- **dxfattribs** (*dict*) – DXF attributes

New in version 0.17.

View Table

```
class ezdxf.sections.table.ViewTable
    Subclass of Table.
```

Collection of [View](#) objects.

add (*name: str*, *, *dxfattribs: Dict[KT, VT] = None*) → View
Add a new view table entry.

Parameters

- **name** (*str*) – view name
- **dxfattribs** (*dict*) – DXF attributes

New in version 0.17.

Viewport Table

class `ezdxf.sections.table.ViewportTable`

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

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

Duplication of table entries is not supported: `duplicate_entry()` raises `NotImplementedError`

add (*name: str*, *, *dxfattribs: Dict[KT, VT] = None*) → VPort

Add a new modelspace viewport entry. A modelspace viewport configuration can consist of multiple viewport entries with the same name.

Parameters

- **name** (*str*) – viewport name, multiple entries possible
- **dxfattribs** (*dict*) – additional DXF attributes

New in version 0.17.

get_config (*self, name: str*) → List[VPort]

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

delete_config (*name: str*) → None

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

Block Record Table

class `ezdxf.sections.table.BlockRecordTable`

Subclass of `Table`.

Collection of `BlockRecord` objects.

add (*name: str*, *, *dxfattribs: Dict[KT, VT] = None*) → BlockRecord

Add a new block record table entry.

Parameters

- **name** (*str*) – block record name
- **dxfattribs** (*dict*) – DXF attributes

New in version 0.17.

Layer

LAYER ([DXF Reference](#)) definition, defines attribute values for entities on this layer for their attributes set to BYLAYER.

Subclass of	<code>ezdxf.entities.DXFEntity</code>
DXF type	'LAYER'
Factory function	<code>Drawing.layers.new()</code>

See also:

[Layer Concept](#) and [Tutorial for Layers](#)

class `ezdxf.entities.Layer`

dxf.handle

DXF handle (feature for experts)

dxf.owner

Handle to owner ([LayerTable](#)).

dxf.name

Layer name, case insensitive and can not contain any of this characters: <>/\":;?*|=` (str)

dxf.flags

Layer flags (bit-coded values, feature for experts)

1	Layer is frozen; otherwise layer is thawed; use <code>is_frozen()</code> , <code>freeze()</code> and <code>thaw()</code>
2	Layer is frozen by default in new viewports
4	Layer is locked; use <code>is_locked()</code> , <code>lock()</code> , <code>unlock()</code>
16	If set, table entry is externally dependent on an xref
32	If both this bit and bit 16 are set, the externally dependent xref has been successfully resolved
64	If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is for the benefit of AutoCAD commands. It can be ignored by most programs that read DXF files and need not be set by programs that write DXF files)

dxf.color

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

dxf.true_color

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

(requires DXF R2004)

dxf.linetype

Name of line type (str)

dxf.plot

Plot flag (int). Whether entities belonging to this layer should be drawn when the document is exported (plotted) to pdf. Does not affect visibility inside the CAD application itself.

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

dx_f.lineweight

Line weight in mm times 100 (e.g. 0.13mm = 13). Smallest line weight is 13 and biggest line weight is 200, values outside this range prevents AutoCAD from loading the file.

ezdxf.lldxf.const.LINEWEIGHT_DEFAULT for using global default line weight.

(requires DXF R13)

dx_f.plotstyle_handle

Handle to plot style name?

(requires DXF R13)

dx_f.material_handle

Handle to default Material.

(requires DXF R13)

rgb

Get/set DXF attribute `dxf.true_color` as (r, g, b) tuple, returns None if attribute `dxf.true_color` is not set.

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

This is the recommend method to get/set RGB values, when ever possible do not use the DXF low level attribute `dxf.true_color`.

color

Get/set layer color, preferred method for getting the layer color, because `dxf.color` is negative for layer status off.

description

Get/set layer description as string

transparency

Get/set layer transparency as float value in the range from 0 to 1. 0 for no transparency (opaque) and 1 for 100% transparency.

is_frozen() → bool

Returns True if layer is frozen.

freeze() → None

Freeze layer.

thaw() → None

Thaw layer.

is_locked() → bool

Returns True if layer is locked.

lock() → None

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

unlock() → None

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

is_off() → bool

Returns True if layer is off.

is_on() → bool

Returns True if layer is on.

on() → NoneSwitch layer *on* (visible).**off()** → NoneSwitch layer *off* (invisible).**get_color()** → intUse property `Layer.color` instead.**set_color(value: int)** → NoneUse property `Layer.color` instead.**rename(name: str)** → None

Rename layer and all known (documented) references to this layer.

Warning: Renaming layers may damage the DXF file in some circumstances!**Parameters** `name` – new layer name**Raises**

- `ValueError` – *name* contains invalid characters: <>/:;?*|=‘
- `ValueError` – layer *name* already exist
- `ValueError` – renaming of layers '0' and 'DEFPOINTS' not possible

Style

Important: DXF is not a layout preserving data format like PDF. It is more similar to the MS Word format. Many applications can open MS Word documents, but the displayed or printed document does not look perfect like the result of MS Word.

The final rendering of DXF files is highly dependent on the interpretation of DXF entities by the rendering engine, and the DXF reference does not provide any guidelines for rendering entities. The biggest visual differences of CAD applications are the text renderings, therefore the only way to get the exact same result is to use the same CAD application.

The DXF format does not and **can not** embed TTF fonts like the PDF format!

The `Textstyle` entity defines a text style ([DXF Reference](#)), and can be used by the entities: `Text`, `Attrib`, `Attdef`, `MText`, `Dimension`, `Leader` and `Multileader`.

Example to create a new text style “Arial” and to apply this text style:

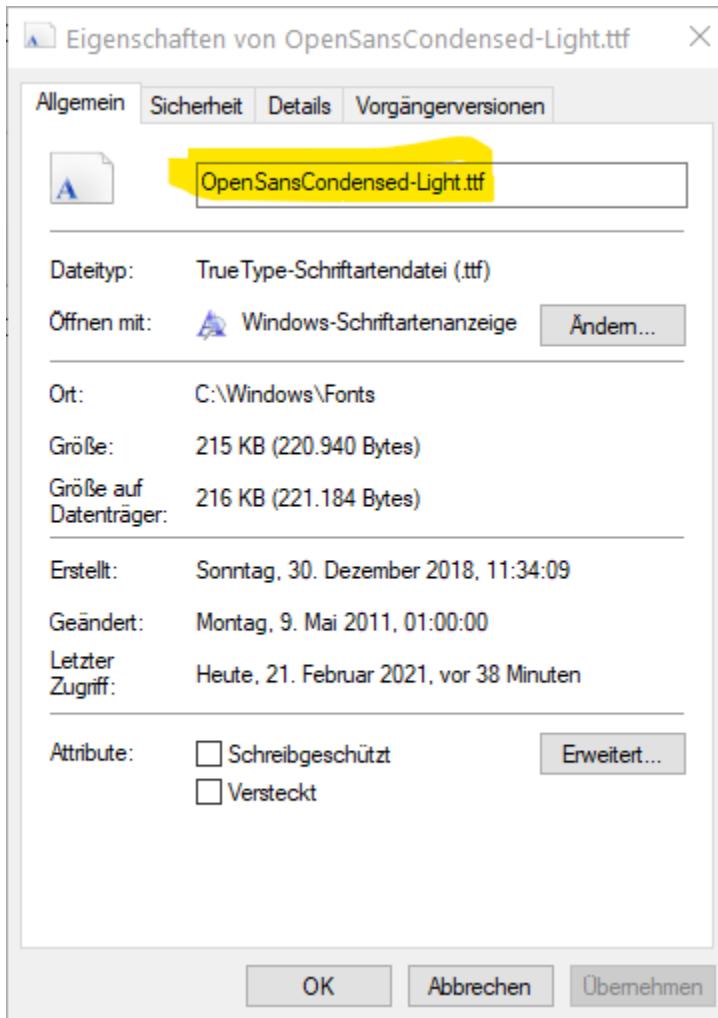
```
doc.styles.add("Arial", font="Arial.ttf")
msp = doc.modelspace()
msp.add_text("my text", dxfattribs={"style": "Arial"})
```

The settings stored in the `Textstyle` entity are the default text style values used by CAD applications if the text settings are not stored in the text entity itself. But not all setting are substituted by the default value. The `height` or `width` attribute must be stored in the text entities itself in order to influence the appearance of the text. It is *recommended* that you do not rely on the default settings in the `Textstyle` entity, set all attributes in the text entity itself if supported.

Font Settings

Just a few settings are available exclusive by the `Textstyle` entity:

The most important setting is the `font` attribute, this attribute defines the rendering font as raw TTF file name, e.g. “Arial.ttf” or “OpenSansCondensed-Light.ttf”, this file name is often **not** the name displayed in GUI application and you have to digg down into the fonts folder e.g. (“C:\Windows\Fonts”) to get the the real file name for the TTF font. Do not include the path!



AutoCAD supports beyond the legacy SHX fonts **only** TTF fonts. The SHX font format is not documented and only available in some CAD applications. The *ezdxf drawing* add-on replaces the SHX fonts by TTF fonts, which look similar to the SHX fonts, unfortunately the license of these fonts is unclear, therefore they can not be packaged with *ezdxf*. They are installed automatically if you use an Autodesk product like *TrueView*, or search the internet at your own risk for these TTF fonts.

The extended font data can provide extra information for the font, it is stored in the XDATA section, not well documented and not widely supported.

Important: The DXF format does not and **can not** embed TTF fonts like the PDF format!

You need to make sure that the CAD application is properly configured to have access to the system fonts. The DXF format has no setting where the CAD application should search for fonts, and does not guarantee that the text rendering

on other computers or operating systems looks the same as on your current system on which you created the DXF.

The second exclusive setting is the vertical text flag in `Textstyle.flags`. The vertical text style is enabled for *all* entities using the text style. Vertical text works only for SHX fonts and is not supported for TTF fonts (in AutoCAD) and is works only for the single line entities `Text` and `Attrib`. Most CAD applications beside AutoCAD and BricsCAD do not support vertical text rendering and even AutoCAD and BricsCAD have problems with vertical text rendering in some circumstances. Using the vertical text feature is not recommended.

Subclass of	<code>ezdxf.entities.DXFEntity</code>
DXF type	'STYLE'
Factory function	<code>Drawing.styles.new()</code>

See also:

Tutorial for Text and DXF internals for *DIMSTYLE Table*.

`class ezdxf.entities.Textstyle`

`dxf.handle`

DXF handle (feature for experts).

`dxf.owner`

Handle to owner (*TextstyleTable*).

`dxf.name`

Style name (str)

`dxf.flags`

Style flags (feature for experts).

1	If set, this entry describes a shape
4	Vertical text
16	If set, table entry is externally dependent on an xref
32	If both this bit and bit 16 are set, the externally dependent xref has been successfully resolved
64	If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is only for the benefit of AutoCAD) commands. It can be ignored by most programs that read DXF files and need not be set by programs that write DXF files)

`dxf.height`

Fixed height in drawing units as float value, 0 for not fixed.

`dxf.width`

Width factor as float value, default value is 1.

`dxf.oblique`

Oblique (slanting) angle in degrees as float value, default value is 0 for no slanting.

`dxf.generation_flags`

Text generations flags as int value.

2	text is backward (mirrored along the x-axis)
4	text is upside down (mirrored about the base line)

`dxf.last_height`

Last height used in drawing units as float value.

dx_f.font

Raw font file name as string without leading path, e.g. “Arial.ttf” for TTF fonts or the SHX font name like “TXT” or “TXT.SHX”.

dx_f.bigfont

Big font name as string, blank if none. No documentation how to use this feature, maybe just a legacy artifact.

get_extended_font_data() → Tuple[str, bool, bool]

Returns extended font data as tuple (font-family, italic-flag, bold-flag).

The extended font data is optional and not reliable! Returns (“”, False, False) if extended font data is not present.

set_extended_font_data(family: str = "", *, italic=False, bold=False) → None

Set extended font data, the font-family name *family* is not validated by *ezdxf*. Overwrites existing data.

discard_extended_font_data()

Discard extended font data.

make_font(cap_height: float = None, width_factor: float = None) → AbstractFont

Returns a font abstraction *AbstractFont* for this text style. Returns a font for a cap height of 1, if the text style has auto height (*Textstyle.dxf.height* is 0) and the given *cap_height* is None or 0. Uses the *Textstyle.dxf.width* attribute if the given *width_factor* is None or 0, the default value is 1. The attribute *Textstyle.dxf.big_font* is ignored.

Linetype

Defines a linetype ([DXF Reference](#)).

Subclass of	<i>ezdxf.entities.DXFEntity</i>
DXF type	'LTYPE'
Factory function	<code>Drawing.linetypes.new()</code>

See also:

[Tutorial for Linetypes](#)

DXF Internals: *LTYPE Table*

class ezdxf.entities.Linetype**dx_f.name**

Linetype name (str).

dx_f.owner

Handle to owner ([Table](#)).

dx_f.description

Linetype description (str).

dx_f.length

Total pattern length in drawing units (float).

dx_f.items

Number of linetype elements (int).

DimStyle

DIMSTYLE (DXF Reference) defines the appearance of *Dimension* entities. Each of this dimension variables starting with "dim..." can be overridden for any *Dimension* entity individually.

Subclass of	<code>ezdxf.entities.DXFEntity</code>
DXF type	'DIMSTYLE'
Factory function	<code>Drawing.dimstyles.new()</code>

```
class ezdxf.entities.DimStyle

dxfs.owner
    Handle to owner (Table).

dxfs.name
    Dimension style name.

dxfs.flags
    Standard flag values (bit-coded values):
```

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

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

dxfs.dimapost
    Prefix/suffix for alternate units dimensions.

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

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

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

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

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

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

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

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

dx_f.dimrnd

Rounding value for decimal dimensions. (default=0)

Rounds all dimensioning distances to the specified value, for instance, if DIMRND is set to 0.25, all distances round to the nearest 0.25 unit. If you set DIMRND to 1.0, all distances round to the nearest integer.

dx_f.dimdle

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

dx_f.dimtp

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

dx_f.dimtm

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

dx_f.dimtxt

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

dx_f.dimcen

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

dx_f.dimtsz

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

dx_f.dimltf

Alternate units dimension scale factor. (default=25.4)

dx_f.dimlfac

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

dx_f.dimtvp

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

dx_f.dimtfac

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

dx_f.dimgap

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

dx_f.dimaltrnd

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

dx_f.dimtol

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

dx_f.dimlim

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

dx_f.dimtih

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

dx_f.dimtoh

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

dx_f.dimse1

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

dx_f.dimse2

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

dx_f.dimtad

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

0	center
1	above
2	outside, handled like above by <i>ezdxf</i>
3	JIS, handled like above by <i>ezdxf</i>
4	below

dx_f.dimzin

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

Values 0-3 affect feet-and-inch dimensions only.

0	Suppresses zero feet and precisely zero inches
1	Includes zero feet and precisely zero inches
2	Includes zero feet and suppresses zero inches
3	Includes zero inches and suppresses zero feet
4	Suppresses leading zeros in decimal dimensions (for example, 0.5000 becomes .5000)
8	Suppresses trailing zeros in decimal dimensions (for example, 12.5000 becomes 12.5)
12	Suppresses both leading and trailing zeros (for example, 0.5000 becomes .5)

dx_f.dimazin

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

0	Displays all leading and trailing zeros
1	Suppresses leading zeros in decimal dimensions (for example, 0.5000 becomes .5000)
2	Suppresses trailing zeros in decimal dimensions (for example, 12.5000 becomes 12.5)
3	Suppresses leading and trailing zeros (for example, 0.5000 becomes .5)

dx_f.dimalt

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

dx_f.dimaltd

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

dx_f.dimtofl

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

dx_f.dimsah

Toggles appearance of arrowhead blocks. (default=0)

dx_f.dimtix

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

dx_f.dimsoxd

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

dx_f.dimclrd

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

dx_f.dimclre

Dimension extension line color. (default=0)

dx_f.dimclrt

Dimension text color. (default=0)

dx_f.dimadec

Controls the number of decimal places for angular dimensions.

dx_f.dimunit

Obsolete, now use DIMLUNIT AND DIMFRAC

dx_f.dimdec

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

dx_f.dimtdec

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

dx_f.dimaltu

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

dx_f.dimaltd

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

dx_f.dimaunit

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

dx_f.dimfrac

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

dx_f.dimlunit

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

dx_f.dimdsep

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

dx_f.dimtmove

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

0	Moves the dimension line with dimension text
1	Adds a leader when dimension text is moved
2	Allows text to be moved freely without a leader

dx_f.dimjust

Horizontal justification of dimension text. (default=0)

0	Center of dimension line
1	Left side of the dimension line, near first extension line
2	Right side of the dimension line, near second extension line
3	Over first extension line
4	Over second extension line

dx_f.dimsd1

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

dx_f.dimsd2

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

dx_f.dimtolj

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

0	Align with bottom line of dimension text
1	Align vertical centered to dimension text
2	Align with top line of dimension text

dx_f.dimtzin
Zero suppression for tolerances values, see [*DimStyle.dxf.dimzin*](#)

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

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

dx_f.dimfit
Obsolete, now use DIMATFIT and DIMTMOVE

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

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

dx_f.dimtxsty
Text style used for dimension text by name.

dx_f.dimtxsty_handle
Text style used for dimension text by handle of STYLE entry. (use [*DimStyle.dxf.dimtxsty*](#) to get/set text style by name)

dx_f.dimldrblk
Specify arrowhead used for leaders by name.

dx_f.dimldrblk_handle
Specify arrowhead used for leaders by handle of referenced block. (use [*DimStyle.dxf.dimldrblk*](#) to get/set arrowhead by name)

dx_f.dimblk_handle
Block type to use for both arrowheads, handle of referenced block. (use [*DimStyle.dxf.dimblk*](#) to get/set arrowheads by name)

dx_f.dimblk1_handle
Block type to use for first arrowhead, handle of referenced block. (use [*DimStyle.dxf.dimblk1*](#) to get/set arrowhead by name)

dx_f.dimblk2_handle
Block type to use for second arrowhead, handle of referenced block. (use [*DimStyle.dxf.dimblk2*](#) to get/set arrowhead by name)

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

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

dx_f.dimltype
Specifies the linetype used for the dimension line as linetype name, requires DXF R2007+

dx_f.dimltype_handle
Specifies the linetype used for the dimension line as handle to LTYPE entry, requires DXF R2007+ (use [*DimStyle.dxf.dimltype*](#) to get/set linetype by name)

dx_f.dimltex1
Specifies the linetype used for the extension line 1 as linetype name, requires DXF R2007+

dx_f.dimlex1_handle

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

dx_f.dimltex2

Specifies the linetype used for the extension line 2 as linetype name, requires DXF R2007+

dx_f.dimlex2_handle

Specifies the linetype used for the extension line 2 as handle to LTYPE entry, requires DXF R2007+ (use *DimStyle.dxf.dimltex2* to get/set linetype by name)

dx_f.dimfxlon

Extension line has fixed length if set to 1, requires DXF R2007+

dx_f.dimfxl

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

dx_f.dimtfill

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

dx_f.dimtfillclr

Text fill custom color as color index (1-255), requires DXF R2007+

dx_f.dimarcssym

Display arc symbol, supported only by *ArcDimension*:

0	arc symbol preceding the measurement text
1	arc symbol above the measurement text
2	disable arc symbol

copy_to_header (dwg: Drawing) → None

Copy all dimension style variables to HEADER section of *doc*.

set_arrows (blk: str = "", blk1: str = "", blk2: str = "", ldrblk: str = "") → None

Set arrows by block names or AutoCAD standard arrow names, set DIMTSZ to 0 which disables tick.

Parameters

- **blk** – block/arrow name for both arrows, if DIMSAH is 0
- **blk1** – block/arrow name for first arrow, if DIMSAH is 1
- **blk2** – block/arrow name for second arrow, if DIMSAH is 1
- **ldrblk** – block/arrow name for leader

set_tick (size: float = 1) → None

Set tick *size*, which also disables arrows, a tick is just an oblique stroke as marker.

Parameters size – arrow size in drawing units**set_text_align (halign: str = None, valign: str = None, vshift: float = None) → None**

Set measurement text alignment, *halign* defines the horizontal alignment (requires DXF R2000+), *valign* defines the vertical alignment, *above1* and *above2* means above extension line 1 or 2 and aligned with extension line.

Parameters

- **halign** – “left”, “right”, “center”, “above1”, “above2”, requires DXF R2000+
- **valign** – “above”, “center”, “below”

- **vshift** – vertical text shift, if *valign* is “center”; >0 shift upward, <0 shift downwards

set_text_format (*prefix*: str = ”, *postfix*: str = ”, *rnd*: float = None, *dec*: int = None, *sep*: str = None, *leading_zeros*: bool = True, *trailing_zeros*: bool = True)

Set dimension text format, like prefix and postfix string, rounding rule and number of decimal places.

Parameters

- **prefix** – Dimension text prefix text as string
- **postfix** – Dimension text postfix text as string
- **rnd** – Rounds all dimensioning distances to the specified value, for instance, if DIMRND is set to 0.25, all distances round to the nearest 0.25 unit. If you set DIMRND to 1.0, all distances round to the nearest integer.
- **dec** – Sets the number of decimal places displayed for the primary units of a dimension, requires DXF R2000+
- **sep** – “.” or “,” as decimal separator, requires DXF R2000+
- **leading_zeros** – Suppress leading zeros for decimal dimensions if False
- **trailing_zeros** – Suppress trailing zeros for decimal dimensions if False

set_dimline_format (*color*: int = None, *linetype*: str = None, *lineweight*: int = None, *extension*: float = None, *disable1*: bool = None, *disable2*: bool = None)

Set dimension line properties

Parameters

- **color** – color index
- **linetype** – linetype as string, requires DXF R2007+
- **lineweight** – line weight as int, 13 = 0.13mm, 200 = 2.00mm, requires DXF R2000+
- **extension** – extension length
- **disable1** – True to suppress first part of dimension line, requires DXF R2000+
- **disable2** – True to suppress second part of dimension line, requires DXF R2000+

set_extline_format (*color*: int = None, *lineweight*: int = None, *extension*: float = None, *offset*: float = None, *fixed_length*: float = None)

Set common extension line attributes.

Parameters

- **color** – color index
- **lineweight** – line weight as int, 13 = 0.13mm, 200 = 2.00mm
- **extension** – extension length above dimension line
- **offset** – offset from measurement point
- **fixed_length** – set fixed length extension line, length below the dimension line

set_extline1 (*linetype*: str = None, *disable*=False)

Set extension line 1 attributes.

Parameters

- **linetype** – linetype for extension line 1, requires DXF R2007+
- **disable** – disable extension line 1 if True

set_extline2 (*linetype: str = None, disable=False*)

Set extension line 2 attributes.

Parameters

- **linetype** – linetype for extension line 2, requires DXF R2007+
- **disable** – disable extension line 2 if True

set_tolerance (*upper: float, lower: float = None, hfactor: float = 1.0, align: ezdxf.enums.MTextLineAlignment = None, dec: int = None, leading_zeros: bool = None, trailing_zeros: bool = None*) → None

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

Parameters

- **upper** – upper tolerance value
- **lower** – lower tolerance value, if None same as upper
- **hfactor** – tolerance text height factor in relation to the dimension text height
- **align** – tolerance text alignment enum `ezdxf.enums.MTextLineAlignment` requires DXF R2000+
- **dec** – Sets the number of decimal places displayed, requires DXF R2000+
- **leading_zeros** – suppress leading zeros for decimal dimensions if False, requires DXF R2000+
- **trailing_zeros** – suppress trailing zeros for decimal dimensions if False, requires DXF R2000+

Changed in version 0.17.2: argument *align* as enum `ezdxf.enums.MTextLineAlignment`

set_limits (*upper: float, lower: float, hfactor: float = 1.0, dec: int = None, leading_zeros: bool = None, trailing_zeros: bool = None*) → None

Set limits text format, upper and lower limit values, text height factor, number of decimal places or leading and trailing zero suppression.

Parameters

- **upper** – upper limit value added to measurement value
- **lower** – lower lower value subtracted from measurement value
- **hfactor** – limit text height factor in relation to the dimension text height
- **dec** – Sets the number of decimal places displayed, requires DXF R2000+
- **leading_zeros** – suppress leading zeros for decimal dimensions if False, requires DXF R2000+
- **trailing_zeros** – suppress trailing zeros for decimal dimensions if False, requires DXF R2000+

VPort

The viewport table (DXF Reference) stores the modelspace viewport configurations. So this entries just modelspace viewports, not paperspace viewports, for paperspace viewports see the `Viewport` entity.

Subclass of	<code>ezdxf.entities.DXFEntity</code>
DXF type	'VPORT'
Factory function	<code>Drawing.viewports.new()</code>

See also:DXF Internals: *VPORT Configuration Table*

class `ezdxf.entities.VPort`
 Subclass of `DXFEntity`

Defines a viewport configurations for the modelspace.

dx.f.owner
 Handle to owner (`ViewportTable`).

dx.f.name
 Viewport name

dx.f.flags
 Standard flag values (bit-coded values):

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

dx.f.lower_left
 Lower-left corner of viewport

dx.f.upper_right
 Upper-right corner of viewport

dx.f.center
 View center point (in `DCS`)

dx.f.snap_base
 Snap base point (in `DCS`)

dx.f.snap_spacing
 Snap spacing X and Y

dx.f.grid_spacing
 Grid spacing X and Y

dx.f.direction_point
 View direction from target point (in `WCS`)

dx.f.target_point
 View target point (in `WCS`)

dx.f.height
 View height

dx.f.aspect_ratio

dx.f.lens_length
 Lens focal length in mm

dx.f.front_clipping
 Front clipping plane (offset from target point)

```
dxf.back_clipping  
    Back clipping plane (offset from target point)  
  
dxf.snap_rotation  
    Snap rotation angle in degrees  
  
dxf.view_twist  
    View twist angle in degrees  
  
dxf.status  
  
dxf.view_mode  
  
dxf.circle_zoom  
  
dxf.fast_zoom  
  
dxf.ucs_icon  
  
dxf.snap_on  
  
dxf.grid_on  
  
dxf.snap_style  
  
dxf.snap_isopair
```

View

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

Subclass of	<code>ezdxf.entities.DXFEntity</code>
DXF type	'VIEW'
Factory function	<code>Drawing.views.new()</code>

See also:

DXF Internals: [VIEW Table](#)

class `ezdxf.entities.View`

```
dxf.owner  
    Handle to owner (Table).  
  
dxf.name  
    Name of view.  
  
dxf.flags  
    Standard flag values (bit-coded values):
```

1	If set, this is a paper space view
16	If set, table entry is externally dependent on an xref
32	If both this bit and bit 16 are set, the externally dependent xref has been successfully resolved
64	If set, the table entry was referenced by at least one entity in the drawing the last time the drawing was edited. (This flag is only for the benefit of AutoCAD)

dx_f.height
View height (in DCS)

dx_f.width
View width (in DCS)

dx_f.center_point
View center point (in DCS)

dx_f.direction_point
View direction from target (in WCS)

dx_f.target_point
Target point (in WCS)

dx_f.lens_length
Lens length

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

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

dx_f.view_twist
Twist angle in degrees.

dx_f.view_mode
View mode (see VIEWMODE system variable)

dx_f.render_mode

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

dx_f.ucs
1 if there is a UCS associated to this view; 0 otherwise

dx_f.ucs_origin
UCS origin as (x, y, z) tuple (appears only if ucs is set to 1)

dx_f.ucs_xaxis
UCS x-axis as (x, y, z) tuple (appears only if ucs is set to 1)

dx_f.ucs_yaxis
UCS y-axis as (x, y, z) tuple (appears only if ucs is set to 1)

dx_f.ucs_ortho_type
Orthographic type of UCS (appears only if ucs is set to 1)

0	UCS is not orthographic
1	Top
2	Bottom
3	Front
4	Back
5	Left
6	Right

dx.f.elevation

UCS elevation

dx.f.ucs_handle

Handle of UCSTable if UCS is a named UCS. If not present, then UCS is unnamed (appears only if `ucs` is set to 1)

dx.f.base_ucs_handle

Handle of UCSTable of base UCS if UCS is orthographic. If not present and `ucs_ortho_type` is non-zero, then base UCS is taken to be WORLD (appears only if `ucs` is set to 1)

dx.f.camera_plottable

1 if the camera is plottable

dx.f.background_handle

Handle to background object (optional)

dx.f.live_selection_handle

Handle to live section object (optional)

dx.f.visual_style_handle

Handle to visual style object (optional)

dx.f.sun_handle

Sun hard ownership handle.

AppID

Defines an APPID ([DXF Reference](#)). These table entries maintain a set of names for all registered applications.

Subclass of	<i>ezdxf.entities.DXFEntity</i>
DXF type	'APPID'
Factory function	Drawing.appids.new()

class ezdxf.entities.AppID**dx.f.owner**

Handle to owner ([Table](#)).

dx.f.name

User-supplied (or application-supplied) application name (for extended data).

dx.f.flags

Standard flag values (bit-coded values):

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

UCS

Defines an named or unnamed user coordinate system ([DXF Reference](#)) for usage in CAD applications. This UCS table entry does not interact with `ezdxf` in any way, to do coordinate transformations by `ezdxf` use the `ezdxf.math.UCS` class.

Subclass of	<code>ezdxf.entities.DXFEntity</code>
DXF type	'UCS'
Factory function	<code>Drawing.ucs.new()</code>

See also:

`UCS` and `OCS`

class `ezdxf.entities.UCSTableEntry`

`dxf.owner`

Handle to owner (`Table`).

`dxf.name`

UCS name (str).

`dxf.flags`

Standard flags (bit-coded values):

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

`dxf.origin`

Origin as (x, y, z) tuple

`dxf.xaxis`

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

`dxf.yaxis`

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

`ucs()` → UCS

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

BlockRecord

BLOCK_RECORD ([DXF Reference](#)) is the core management structure for `BlockLayout` and `Layout`. This is an internal DXF structure managed by `ezdxf`, package users don't have to care about it.

Subclass of	<code>ezdxf.entities.DXFentity</code>
DXF type	'BLOCK_RECORD'
Factory function	<code>Drawing.block_records.new()</code>

```
class ezdxf.entities.BlockRecord
```

dx.f.owner

Handle to owner ([Table](#)).

dx.f.name

Name of associated BLOCK.

dx.f.layout

Handle to associated DXFLayout, if paperspace layout or modelspace else "0"

dx.f.explode

1 for BLOCK references can be exploded else 0

dx.f.scale

1 for BLOCK references can be scaled else 0

dx.f.units

BLOCK insert units

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

Internal Structure

Do not change this structures, this is just an information for experienced developers!

The BLOCK_RECORD is the owner of all the entities in a layout and stores them in an [EntitySpace](#) object (`BlockRecord.entity_space`). For each layout exist a BLOCK definition in the BLOCKS section, a reference to the [Block](#) entity is stored in `BlockRecord.block`.

[Modelspace](#) and [Paperspace](#) layouts require an additional [DXFLayout](#) object in the OBJECTS section.

See also:

More information about [Block Management Structures](#) and [Layout Management Structures](#).

Blocks

A block definition ([BlockLayout](#)) is a collection of DXF entities, which can be placed multiply times at different layouts or other blocks as references to the block definition.

See also:

[Tutorial for Blocks](#) and DXF Internals: [Block Management Structures](#)

Block

BLOCK ([DXF Reference](#)) entity is embedded into the [BlockLayout](#) object. The BLOCK entity is accessible by the `BlockLayout.block` attribute.

Subclass of	<code>ezdxf.entities.DXFEntity</code>
DXF type	'BLOCK'
Factory function	<code>Drawing.blocks.new()</code> (returns a BlockLayout)

See also:

[Tutorial for Blocks](#) and DXF Internals: [Block Management Structures](#)

`class ezdxf.entities.Block`

`dxfl.handle`
BLOCK handle as plain hex string. (feature for experts)

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

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

`dxfl.name`
BLOCK name as string. (case insensitive)

`dxfl.base_point`
BLOCK base point as (x, y, z) tuple, default value is (0, 0, 0)

Insertion location referenced by the [Insert](#) entity to place the block reference and also the center of rotation and scaling.

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

1	Anonymous block generated by hatching, associative dimensioning, other internal operations, or an application
2	Block has non-constant attribute definitions (this bit is not set if the block has any attribute definitions that are constant, or has no attribute definitions at all)
4	Block is an external reference (xref)
8	Block is an xref overlay
16	Block is externally dependent
32	This is a resolved external reference, or dependent of an external reference (ignored on input)
64	This definition is a referenced external reference (ignored on input)

dx.f.xref_path

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

is_layout_block

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

is_anonymous

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

is_xref

Returns True if block is an external referenced file.

is_xref_overlay

Returns True if block is an external referenced overlay file.

EndBlk

ENDBLK entity is embedded into the *BlockLayout* object. The ENDBLK entity is accessible by the *BlockLayout.endblk* attribute.

Subclass of	<i>ezdxf.entities.DXFEntity</i>
DXF type	'ENDBLK'

class ezdxf.entities.EndBlk**dx.f.handle**

BLOCK handle as plain hex string. (feature for experts)

dx.f.owner

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

dx.f.layer

Layer name as string; should always be the same as *Block.dx.f.layer*

Insert

Block reference (DXF Reference) with maybe attached attributes (*Attrib*).

Subclass of	<i>ezdxf.entities.DXFGraphic</i>
DXF type	'INSERT'
Factory function	<i>ezdxf.layouts.BaseLayout.add_blockref()</i>
Inherited DXF attributes	<i>Common graphical DXF attributes</i>

See also:

[Tutorial for Blocks](#)

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

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

class ezdxf.entities.**Insert**

dx_f.name

BLOCK name (str)

dx_f.insert

Insertion location of the BLOCK base point as (2D/3D Point in *OCS*)

dx_f.xscale

Scale factor for x direction (float)

dx_f.yscale

Scale factor for y direction (float)

Not all CAD applications support non-uniform scaling (e.g. LibreCAD).

dx_f.zscale

Scale factor for z direction (float)

Not all CAD applications support non-uniform scaling (e.g. LibreCAD).

dx_f.rotation

Rotation angle in degrees (float)

dx_f.row_count

Count of repeated insertions in row direction, MININSERT entity if > 1 (int)

dx_f.row_spacing

Distance between two insert points (MININSERT) in row direction (float)

dx_f.column_count

Count of repeated insertions in column direction, MININSERT entity if > 1 (int)

dx_f.column_spacing

Distance between two insert points (MININSERT) in column direction (float)

attribs

A list of all attached *Attrib* entities.

has_scaling

Returns True if any axis scaling is applied.

has_uniform_scaling

Returns True if scaling is uniform in x-, y- and z-axis ignoring reflections e.g. (1, 1, -1) is uniform scaling.

mcount

Returns the multi-insert count, MININSERT (multi-insert) processing is required if *mcount* > 1.

set_scale (*factor*: float)

Set uniform scaling.

block() → Optional[*BlockLayout*]

Returns associated *BlockLayout*.

place (*insert: Vertex = None, scale: Tuple[float, float, float] = None, rotation: float = None*) → Insert
Set block reference placing location *insert*, scaling and rotation attributes. Parameters which are *None* will not be altered.

Parameters

- **insert** – insert location as (x, y [, z]) tuple
- **scale** – (x-scale, y-scale, z-scale) tuple
- **rotation** – rotation angle in degrees

grid (*size: Tuple[int, int] = (1, 1), spacing: Tuple[float, float] = (1, 1)*) → Insert
Place block reference in a grid layout, grid *size* defines the row- and column count, *spacing* defines the distance between two block references.

Parameters

- **size** – grid size as (row_count, column_count) tuple
- **spacing** – distance between placing as (row_spacing, column_spacing) tuple

has_attrib (*tag: str, search_const: bool = False*) → bool
Returns True if ATTRIB *tag* exist, for *search_const* doc see [get_attrib\(\)](#).

Parameters

- **tag** – tag name as string
- **search_const** – search also const ATTDEF entities

get_attrib (*tag: str, search_const: bool = False*) → Union[Attrib, AttDef, None]
Get attached *Attrib* entity with *dxf.tag == tag*, returns *None* if not found. Some applications may not attach constant ATTRIB entities, set *search_const* to True, to get at least the associated *AttDef* entity.

Parameters

- **tag** – tag name
- **search_const** – search also const ATTDEF entities

get_attrib_text (*tag: str, default: str = "", search_const: bool = False*) → str
Get content text of attached *Attrib* entity with *dxf.tag == tag*, returns *default* if not found. Some applications may not attach constant ATTRIB entities, set *search_const* to True, to get content text of the associated *AttDef* entity.

Parameters

- **tag** – tag name
- **default** – default value if ATTRIB *tag* is absent
- **search_const** – search also const ATTDEF entities

add_attrib (*tag: str, text: str, insert: Vertex = (0, 0), dxفاتtribs=None*) → Attrib
Attach an *Attrib* entity to the block reference.

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

```
e.add_attrib('EXAMPLETAG', 'example text').set_placement(  
    (3, 7), align=TextEntityAlignment.MIDDLE_CENTER  
)
```

Parameters

- **tag** – tag name as string
- **text** – content text as string
- **insert** – insert location as tuple (`x`, `y` [, `z`]) in *WCS*
- **dxfattribs** – additional DXF attributes for the ATTRIB entity

add_auto_attribs (`values: Dict[str, str]`) → ezdxf.entities.insert.Insert

Attach for each `Attdef` entity, defined in the block definition, automatically an `Attrib` entity to the block reference and set tag/value DXF attributes of the ATTRIB entities by the key/value pairs (both as strings) of the `values` dict. The ATTRIB entities are placed relative to the insert location of the block reference, which is identical to the block base point.

This method avoids the wrapper block of the `add_auto_blockref()` method, but the visual results may not match the results of CAD applications, especially for non uniform scaling. If the visual result is very important to you, use the `add_auto_blockref()` method.

Parameters `values` – `Attrib` tag values as tag/value pairs

delete_attrib (`tag: str, ignore=False`) → None

Delete an attached `Attrib` entity from INSERT. If `ignore` is `False`, an `DXFKeyError` exception is raised, if ATTRIB `tag` does not exist.

Parameters

- **tag** – ATTRIB name
- **ignore** – `False` for raising `DXFKeyError` if ATTRIB `tag` does not exist.

Raises `DXFKeyError` – if ATTRIB `tag` does not exist.

delete_all_attribs () → None

Delete all `Attrib` entities attached to the INSERT entity.

reset_transformation () → None

Reset block reference parameters *location*, *rotation* and *extrusion* vector.

transform (`m: Matrix44`) → Insert

Transform INSERT entity by transformation matrix `m` inplace.

Unlike the transformation matrix `m`, the INSERT entity can not represent a non orthogonal target coordinate system, for this case an `InsertTransformationError` will be raised.

translate (`dx: float, dy: float, dz: float`) → Insert

Optimized INSERT translation about `dx` in x-axis, `dy` in y-axis and `dz` in z-axis.

virtual_entities (`skipped_entity_callback: Callable[[DXFGraphic, str], None] = None`) → Iterable[DXFGraphic]

Yields “virtual” entities of a block reference. This method is meant to examine the block reference entities at the “exploded” location without really “exploding” the block reference. The ‘`skipped_entity_callback()`’ will be called for all entities which are not processed, signature: `skipped_entity_callback(entity: DXFEntity, reason: str)`, `entity` is the original (untransformed) DXF entity of the block definition, the `reason` string is an explanation why the entity was skipped.

This entities are not stored in the entity database, have no handle and are not assigned to any layout. It is possible to convert this entities into regular drawing entities by adding the entities to the entities database and a layout of the same DXF document as the block reference:

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

This method does not resolve the MININSERT attributes, only the sub-entities of the base INSERT will be returned. To resolve MININSERT entities check if multi insert processing is required, that's the case if property `Insert.mcount` > 1, use the `Insert.multi_insert()` method to resolve the MININSERT entity into single INSERT entities.

Warning: Non uniform scaling may return incorrect results for text entities (TEXT, MTEXT, ATTRIB) and maybe some other entities.

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

`multi_insert() → Iterable[Insert]`

Yields a virtual INSERT entity for each grid element of a MININSERT entity (multi-insert).

`explode(target_layout: BaseLayout = None) → EntityQuery`

Explode block reference entities into target layout, if target layout is `None`, the target layout is the layout of the block reference. This method destroys the source block reference entity.

Transforms the block entities into the required `WCS` location by applying the block reference attributes `insert`, `extrusion`, `rotation` and the scaling values `xscale`, `yscale` and `zscale`.

Attached ATTRIB entities are converted to TEXT entities, this is the behavior of the BURST command of the AutoCAD Express Tools.

Returns an `EntityQuery` container with all “exploded” DXF entities.

Warning: Non uniform scaling may lead to incorrect results for text entities (TEXT, MTEXT, ATTRIB) and maybe some other entities.

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

`ucs() → UCS`

Returns the block reference coordinate system as `ezdxf.math.ucs` object.

Atrib

The ATTRIB (DXF Reference) entity represents a text value associated with a tag. In most cases an ATTRIB is appended to an `Insert` entity, but it can also appear as standalone entity.

Subclass of	<code>ezdxf.entities.Text</code>
DXF type	'ATTRIB'
Factory function	<code>ezdxf.layouts.BaseLayout.add_attrib()</code> (stand alone entity)
Factory function	<code>Insert.add_attrib()</code> (attached to <code>Insert</code>)
Inherited DXF attributes	<code>Common graphical DXF attributes</code>

See also:

[Tutorial for Blocks](#)

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

`class ezdxf.entities.Attrib`

ATTRIB supports all DXF attributes and methods of parent class `Text`.

`dxfs.tag`

Tag to identify the attribute (str)

`dxfs.text`

Attribute content as text (str)

`virtual_mtext_entity() → ezdxf.entities.mtext.MText`

Returns the embedded MTEXT entity as a regular but virtual `MText` entity with the same graphical properties as the host entity.

`plain_mtext(fast=True) → str`

Returns the embedded MTEXT content without formatting codes. Returns an empty string if no embedded MTEXT entity exist.

The “fast” mode is accurate if the DXF content was created by reliable (and newer) CAD applications like AutoCAD or BricsCAD. The “accurate” mode is for some rare cases where the content was created by older CAD applications or unreliable DXF libraries and CAD applications.

The “accurate” mode is **much** slower than the “fast” mode.

Parameters `fast` – uses the “fast” mode to extract the plain MTEXT content if `True` or the “accurate” mode if set to `False`

`set_mtext(mtext: ezdxf.entities.mtext.MText, graphic_properties=True) → None`

Set multi-line properties from a `MText` entity.

The multi-line ATTRIB/ATTDEF entity requires DXF R2018, otherwise an ordinary single line ATTRIB/ATTDEF entity will be exported.

Parameters

- `mtext` – source `MText` entity
- `graphic_properties` – copy graphic properties (color, layer, ...) from source MTEXT if `True`

`embed_mtext(mtext: ezdxf.entities.mtext.MText, graphic_properties=True) → None`

Set multi-line properties from a `MText` entity and destroy the source entity afterwards.

The multi-line ATTRIB/ATTDEF entity requires DXF R2018, otherwise an ordinary single line ATTRIB/ATTDEF entity will be exported.

Parameters

- `mtext` – source `MText` entity
- `graphic_properties` – copy graphic properties (color, layer, ...) from source MTEXT if `True`

AttDef

The ATTDEF (DXF Reference) entity is a template in a `BlockLayout`, which will be used to create an attached `Attrib` entity for an `Insert` entity.

Subclass of	<code>ezdxf.entities.Text</code>
DXF type	'ATTDEF'
Factory function	<code>ezdxf.layouts.BaseLayout.add_attdef()</code>
Inherited DXF attributes	<i>Common graphical DXF attributes</i>

See also:

[Tutorial for Blocks](#)

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

class `ezdxf.entities.AttDef`

ATTDEF supports all DXF attributes and methods of parent class `Text`.

dx.dxf.tag

Tag to identify the attribute (str)

dx.dxf.text

Attribute content as text (str)

dx.dxf.prompt

Attribute prompt string. (CAD application feature)

dx.dxf.field_length

Just relevant to CAD programs for validating user input

virtual_mtext_entity () → ezdxf.entities.mtext.MText

Returns the embedded MTEXT entity as a regular but virtual `MText` entity with the same graphical properties as the host entity.

plain_mtext (fast=True) → str

Returns the embedded MTEXT content without formatting codes. Returns an empty string if no embedded MTEXT entity exist.

The “fast” mode is accurate if the DXF content was created by reliable (and newer) CAD applications like AutoCAD or BricsCAD. The “accurate” mode is for some rare cases where the content was created by older CAD applications or unreliable DXF libraries and CAD applications.

The “accurate” mode is **much** slower than the “fast” mode.

Parameters `fast` – uses the “fast” mode to extract the plain MTEXT content if `True` or the “accurate” mode if set to `False`

set_mtext (mtext: ezdxf.entities.mtext.MText, graphic_properties=True) → None

Set multi-line properties from a `MText` entity.

The multi-line ATTRIB/ATTDEF entity requires DXF R2018, otherwise an ordinary single line ATTRIB/ATTDEF entity will be exported.

Parameters

- `mtext` – source `MText` entity
- `graphic_properties` – copy graphic properties (color, layer, ...) from source MTEXT if `True`

embed_mtext (mtext: ezdxf.entities.mtext.MText, graphic_properties=True) → None

Set multi-line properties from a `MText` entity and destroy the source entity afterwards.

The multi-line ATTRIB/ATTDEF entity requires DXF R2018, otherwise an ordinary single line ATTRIB/ATTDEF entity will be exported.

Parameters

- **mtext** – source `MText` entity
- **graphic_properties** – copy graphic properties (color, layer, ...) from source MTEXT if True

Layouts

Layout Manager

The layout manager is unique to each DXF drawing, access the layout manager as `layouts` attribute of the `Drawing` object (e.g. `doc.layouts.rename("Layout1", "PlanView")`).

`class ezdxf.layouts.Layouts`

The `Layouts` class manages `Paperspace` layouts and the `Modelspace`.

`__len__() → int`

Returns count of existing layouts, including the modelspace layout.

`__contains__(name: str) → bool`

Returns True if layout `name` exist.

`__iter__() → Iterable[Layout]`

Returns iterable of all layouts as `Layout` objects, including the modelspace layout.

`names() → List[str]`

Returns a list of all layout names, all names in original case sensitive form.

`names_in_taborder() → List[str]`

Returns all layout names in tab order as shown in `CAD` applications.

`modelspace() → Modelspace`

Returns the `Modelspace` layout.

`get(name: str) → Layout`

Returns `Layout` by `name`, case insensitive “Model” == “MODEL”.

Parameters `name` – layout name as shown in tab, e.g. ‘Model’ for modelspace

`new(name: str, dxfattribs: dict = None) → Paperspace`

Returns a new `Paperspace` layout.

Parameters

- **name** – layout name as shown in tabs in `CAD` applications
- **dfxattribs** – additional DXF attributes for the `DXFLayout` entity

Raises

- `DXFValueError` – Invalid characters in layout name.

- `DXFValueError` – Layout `name` already exist.

`rename(old_name: str, new_name: str) → None`

Rename a layout from `old_name` to `new_name`. Can not rename layout ‘Model’ and the new name of a layout must not exist.

Parameters

- **old_name** – actual layout name, case insensitive
- **new_name** – new layout name, case insensitive

Raises

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

delete(*name*: str) → NoneDelete layout *name* and destroy all entities in that layout.**Parameters** **name** (str) – layout name as shown in tabs**Raises**

- DXFKeyError – if layout *name* do not exists
- DXFValueError – deleting modelspace layout is not possible
- DXFValueError – deleting last paperspace layout is not possible

active_layout() → Paperspace

Returns the active paperspace layout.

set_active_layout(*name*: str) → NoneSet layout *name* as active paperspace layout.**get_layout_for_entity**(*entity*: DXFEntity) → LayoutReturns the owner layout for a DXF *entity*.

Layout Types

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

- *Modelspace* is the common working space, containing basic drawing entities.
- *Paperspace* is arrangement of objects for printing and plotting, this layout contains basic drawing entities and viewports to the *Modelspace*.
- *BlockLayout* works on an associated Block, Blocks are collections of drawing entities for reusing by block references.

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

Entity Ownership

A layout owns all entities residing in their entity space, this means the `dx.dxf.owner` attribute of any `DXFGraphic` in this layout is the `dx.dxf.handle` of the layout, and deleting an entity from a layout is the end of life of this entity, because it is also deleted from the `EntityDB`. But it is possible to just unlink an entity from a layout, so it can be assigned to another layout, use the `move_to_layout()` method to move entities between layouts.

BaseLayout

class ezdxf.layouts.**BaseLayout**

BaseLayout is the common base class for *Layout* and *BlockLayout*.

is_alive
False if layout is deleted.

is_active_paperspace
True if is active layout.

is_any_paperspace
True if is any kind of paperspace layout.

is_modelspace
True if is modelspace layout.

is_any_layout
True if is any kind of modelspace or paperspace layout.

is_block_layout
True if not any kind of modelspace or paperspace layout, just a regular block definition.

units
set drawing units.

Type Get/Set layout/block drawing units as enum, see also

Type ref

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

__iter__ () → Iterator[DXFGraphic]
Returns iterable of all drawing entities in this layout.

__getitem__ (index)
Get entity at *index*.

The underlying data structure for storing entities is organized like a standard Python list, therefore *index* can be any valid list indexing or slicing term, like a single index `layout[-1]` to get the last entity, or an index slice `layout[:10]` to get the first 10 or less entities as `List[DXFGraphic]`.

get_extension_dict () → ExtensionDict
Returns the associated extension dictionary, creates a new one if necessary.

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

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

unlink_entity (*entity*: DXFGraphic) → None
Unlink *entity* from layout but does not delete entity from the entity database, this removes *entity* just from the layout entity space.

purge ()
Remove all destroyed entities from the layout entity space.

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

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

Parameters

- **dxfattrib** – grouping by DXF attribute like 'layer'

- **key** – key function, which accepts a `DXFGraphic` entity as argument and returns the grouping key of an entity or `None` to ignore the entity. Reason for ignoring: a queried DXF attribute is not supported by entity.

`move_to_layout(entity: DXFGraphic, layout: BaseLayout) → None`

Move entity to another layout.

Parameters

- **entity** – DXF entity to move
- **layout** – any layout (modelspace, paperspace, block) from **same** drawing

`add_entity(entity: DXFGraphic) → None`

Add an existing `DXFGraphic` entity to a layout, but be sure to unlink (`unlink_entity()`) entity from the previous owner layout. Adding entities from a different DXF drawing is not supported.

`add_foreign_entity(entity: DXFGraphic, copy=True) → None`

Add a foreign DXF entity to a layout, this foreign entity could be from another DXF document or an entity without an assigned DXF document. The intention of this method is to add **simple** entities from another DXF document or from a DXF iterator, for more complex operations use the `importer` add-on. Especially objects with BLOCK section (INSERT, DIMENSION, MLEADER) or OBJECTS section dependencies (IMAGE, UNDERLAY) can not be supported by this simple method.

Not all DXF types are supported and every dependency or resource reference from another DXF document will be removed except attribute layer will be preserved but only with default attributes like color 7 and linetype CONTINUOUS because the layer attribute doesn't need a layer table entry.

If the entity is part of another DXF document, it will be unlinked from this document and its entity database if argument `copy` is `False`, else the entity will be copied. Unassigned entities like from DXF iterators will just be added.

Supported DXF types:

- POINT
- LINE
- CIRCLE
- ARC
- ELLIPSE
- LWPOLYLINE
- SPLINE
- POLYLINE
- 3DFACE
- SOLID
- TRACE
- SHAPE
- MESH
- ATTRIB
- ATTDEF
- TEXT
- MTEXT

- HATCH

Parameters

- **entity** – DXF entity to copy or move
- **copy** – if True copy entity from other document else unlink from other document

add_point (*location: Vertex, dxfattribs=None*) → Point

Add a *Point* entity at *location*.

Parameters

- **location** – 2D/3D point in *WCS*
- **dxfattribs** – additional DXF attributes

add_line (*start: Vertex, end: Vertex, dxfattribs=None*) → Line

Add a *Line* entity from *start* to *end*.

Parameters

- **start** – 2D/3D point in *WCS*
- **end** – 2D/3D point in *WCS*
- **dxfattribs** – additional DXF attributes

add_circle (*center: Vertex, radius: float, dxfattribs=None*) → Circle

Add a *Circle* entity. This is an 2D element, which can be placed in space by using *OCS*.

Parameters

- **center** – 2D/3D point in *WCS*
- **radius** – circle radius
- **dxfattribs** – additional DXF attributes

add_ellipse (*center: Vertex, major_axis: Vertex = (1, 0, 0), ratio: float = 1, start_param: float = 0, end_param: float = 6.283185307179586, dxfattribs=None*) → Ellipse

Add an *Ellipse* entity, *ratio* is the ratio of minor axis to major axis, *start_param* and *end_param* defines start and end point of the ellipse, a full ellipse goes from 0 to 2π . The ellipse goes from start to end param in *counter clockwise* direction.

Parameters

- **center** – center of ellipse as 2D/3D point in *WCS*
- **major_axis** – major axis as vector (x, y, z)
- **ratio** – ratio of minor axis to major axis in range $[-1e-6, 1.0]$
- **start_param** – start of ellipse curve
- **end_param** – end param of ellipse curve
- **dxfattribs** – additional DXF attributes

add_arc (*center: Vertex, radius: float, start_angle: float, end_angle: float, is_counter_clockwise: bool = True, dxfattribs=None*) → Arc

Add an *Arc* entity. The arc goes from *start_angle* to *end_angle* in counter clockwise direction by default, set parameter *is_counter_clockwise* to False for clockwise orientation.

Parameters

- **center** – center of arc as 2D/3D point in *WCS*

- **radius** – arc radius
- **start_angle** – start angle in degrees
- **end_angle** – end angle in degrees
- **is_counter_clockwise** – False for clockwise orientation
- **dxfattribs** – additional DXF attributes

add_solid (*points*: *Iterable[Vertex]*, *dxfattribs=None*) → Solid

Add a *Solid* entity, *points* is an iterable of 3 or 4 points.

Hint: The last two vertices are in reversed order: a square has the vertex order 0-1-3-2

Parameters

- **points** – iterable of 3 or 4 2D/3D points in *WCS*
- **dxfattribs** – additional DXF attributes

add_trace (*points*: *Iterable[Vertex]*, *dxfattribs=None*) → Trace

Add a *Trace* entity, *points* is an iterable of 3 or 4 points.

Hint: The last two vertices are in reversed order: a square has the vertex order 0-1-3-2

Parameters

- **points** – iterable of 3 or 4 2D/3D points in *WCS*
- **dxfattribs** – additional DXF attributes

add_3dface (*points*: *Iterable[Vertex]*, *dxfattribs=None*) → Face3d

Add a *3DFace* entity, *points* is an iterable 3 or 4 2D/3D points.

Hint: In contrast to SOLID and TRACE, the last two vertices are in regular order: a square has the vertex order 0-1-2-3

Parameters

- **points** – iterable of 3 or 4 2D/3D points in *WCS*
- **dxfattribs** – additional DXF attributes

add_text (*text*: str, *dxfattribs=None*) → Text

Add a *Text* entity, see also *Textstyle*.

Parameters

- **text** – content string
- **dxfattribs** – additional DXF attributes

add_blockref (*name*: str, *insert*: Vertex, *dxfattribs=None*) → Insert

Add an *Insert* entity.

When inserting a block reference into the modelspace or another block layout with different units, the scaling factor between these units should be applied as scaling attributes (`xscale`, ...) e.g. modelspace in meters and block in centimeters, `xscale` has to be 0.01.

Parameters

- `name` – block name as str
- `insert` – insert location as 2D/3D point in `WCS`
- `dxfattribs` – additional DXF attributes

`add_auto_blockref(name: str, insert: Vertex, values: Dict[str, str], dxfattribs=None) → Insert`

Add an `Insert` entity. This method adds for each `Attdef` entity, defined in the block definition, automatically an `Attrib` entity to the block reference and set (tag, value) DXF attributes of the `ATTRIB` entities by the (key, value) pairs (both as strings) of the `values` dict.

The `Attrib` entities are placed relative to the insert point, which is equal to the block base point.

This method wraps the `INSERT` and all the `ATTRIB` entities into an anonymous block, which produces the best visual results, especially for non uniform scaled block references, because the transformation and scaling is done by the CAD application. But this makes evaluation of block references with attributes more complicated, if you prefer `INSERT` and `ATTRIB` entities without a wrapper block use the `add_blockref_with_attribs()` method.

Parameters

- `name` – block name
- `insert` – insert location as 2D/3D point in `WCS`
- `values` – `Attrib` tag values as (tag, value) pairs
- `dxfattribs` – additional DXF attributes

`add_attdef(tag: str, insert: Vertex = (0, 0), text: str = "", dxfattribs=None) → AttDef`

Add an `AttDef` as stand alone DXF entity.

Set position and alignment by the idiom:

```
layout.add_attdef("NAME").set_placement(
    (2, 3), align=TextEntityAlignment.MIDDLE_CENTER
)
```

Parameters

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

`add_polyline2d(points: Iterable[Vertex], format: str = None, *, close: bool = False, dxfattribs=None) → Polyline`

Add a 2D `Polyline` entity.

Parameters

- `points` – iterable of 2D points in `WCS`
- `close` – `True` for a closed polyline
- `format` – user defined point format like `add_lwpolyline()`, default is `None`

- **dxfattribs** – additional DXF attributes

add polyline3d(*points*: Iterable[Vertex], *, *close* = False, *dfattribs*=None) → Polyline
Add a 3D [Polyline](#) entity.

Parameters

- **points** – iterable of 3D points in [WCS](#)
- **close** – True for a closed polyline
- **dfattribs** – additional DXF attributes

add polymesh(*size*: Tuple[int, int] = (3, 3), *dfattribs*=None) → Polymesh

Add a [Polymesh](#) entity, which is a wrapper class for the POLYLINE entity. A polymesh is a grid of *mcount* x *ncount* vertices and every vertex has its own (x, y, z)-coordinates.

Parameters

- **size** – 2-tuple (*mcount*, *ncount*)
- **dfattribs** – additional DXF attributes

add polyface(*dfattribs*=None) → Polyface

Add a [Polyface](#) entity, which is a wrapper class for the POLYLINE entity.

Parameters **dfattribs** – additional DXF attributes for [Polyline](#) entity

add shape(*name*: str, *insert*: Vertex = (0, 0), *size*: float = 1.0, *dfattribs*=None) → Shape

Add a [Shape](#) reference to a external stored shape.

Parameters

- **name** – shape name as string
- **insert** – insert location as 2D/3D point in [WCS](#)
- **size** – size factor
- **dfattribs** – additional DXF attributes

add lwpolyline(*points*: Iterable[Vertex], *format*: str = 'xyseb', *, *close* = False, *dfattribs*=None) → LWPolyline

Add a 2D polyline as [LWPolyline](#) entity. A points are defined as (x, y, [start_width, [end_width, [bulge]]]) tuples, but order can be redefined by the *format* argument. Set *start_width*, *end_width* to 0 to be ignored like (x, y, 0, 0, bulge).

The [LWPolyline](#) is defined as a single DXF entity and needs less disk space than a [Polyline](#) entity.
(requires DXF R2000)

Format codes:

- x = x-coordinate
- y = y-coordinate
- s = start width
- e = end width
- b = bulge value
- v = (x, y [,z]) tuple (z-axis is ignored)

Parameters

- **points** – iterable of (x, y, [start_width, [end_width, [bulge]]]) tuples

- **format** – user defined point format, default is “xyseb”
- **close** – True for a closed polyline
- **dxfattribs** – additional DXF attributes

add_mtext (*text: str, dxfattribs=None*) → MText

Add a multiline text entity with automatic text wrapping at boundaries as *MText* entity. (requires DXF R2000)

Parameters

- **text** – content string
- **dxfattribs** – additional DXF attributes

add_mtext_static_columns (*content: Iterable[str], width: float, gutter_width: float, height: float, dxfattribs=None*) → MText

Add a multiline text entity with static columns as *MText* entity. The content is spread across the columns, the count of content strings determine the count of columns.

This factory method adds automatically a column break "\N" at the end of each column text to force a new column. The *height* attribute should be big enough to reserve enough space for the tallest column. Too small values produce valid DXF files, but the visual result will not be as expected. The *height* attribute also defines the total height of the MTEXT entity.

(requires DXF R2000)

Parameters

- **content** – iterable of column content
- **width** – column width
- **gutter_width** – distance between columns
- **height** – max. column height
- **dxfattribs** – additional DXF attributes

New in version 0.17.

add_mtext_dynamic_manual_height_columns (*content: str, width: float, gutter_width: float, heights: Sequence[float], dxfattribs=None*) → MText

Add a multiline text entity with dynamic columns as *MText* entity. The content is spread across the columns automatically by the CAD application. The *heights* sequence determine the height of the columns, except for the last column, which always takes the remaining content. The height value for the last column is required but can be 0, because the value is ignored. The count of *heights* also determines the count of columns, and *max(heights)* defines the total height of the MTEXT entity, which may be wrong if the last column requires more space.

This current implementation works best for DXF R2018, because the content is stored as a continuous text in a single MTEXT entity. For DXF versions prior to R2018 the content should be distributed across multiple MTEXT entities (one entity per column), which is not done by *ezdxf*, but the result is correct for advanced DXF viewers and CAD application, which do the MTEXT content distribution completely by itself.

(requires DXF R2000)

Parameters

- **content** – column content as a single string
- **width** – column width

- **gutter_width** – distance between columns
- **heights** – column height for each column
- **dxfattribs** – additional DXF attributes

New in version 0.17.

add_mtext_dynamic_auto_height_columns (*content: str, width: float, gutter_width: float, height: float, count: int, dxfattribs=None*) → MText

Add a multiline text entity with as much columns as needed for the given common fixed *height*. The content is spread across the columns automatically by the CAD application. The *height* argument also defines the total height of the MTEXT entity. To get the correct column *count* requires an **exact** MTEXT rendering like AutoCAD, which is not done by *ezdxf*, therefore passing the expected column *count* is required to calculate the correct total width.

This current implementation works best for DXF R2018, because the content is stored as a continuous text in a single MTEXT entity. For DXF versions prior to R2018 the content should be distributed across multiple MTEXT entities (one entity per column), which is not done by *ezdxf*, but the result is correct for advanced DXF viewers and CAD application, which do the MTEXT content distribution completely by itself.

Because of the current limitations the use of this method is not recommended. This situation may improve in future releases, but the exact rendering of the content will also slow down the processing speed dramatically.

(requires DXF R2000)

Parameters

- **content** – column content as a single string
- **width** – column width
- **gutter_width** – distance between columns
- **height** – max. column height
- **count** – expected column count
- **dxfattribs** – additional DXF attributes

New in version 0.17.

add_ray (*start: Vertex, unit_vector: Vertex, dxfattribs=None*) → Ray

Add a [Ray](#) that begins at *start* point and continues to infinity (construction line). (requires DXF R2000)

Parameters

- **start** – location 3D point in [WCS](#)
- **unit_vector** – 3D vector (x, y, z)
- **dxfattribs** – additional DXF attributes

add_xline (*start: Vertex, unit_vector: Vertex, dxfattribs=None*) → XLine

Add an infinity [XLine](#) (construction line). (requires DXF R2000)

Parameters

- **start** – location 3D point in [WCS](#)
- **unit_vector** – 3D vector (x, y, z)
- **dxfattribs** – additional DXF attributes

`add_mline(vertices: Iterable[Vertex] = None, *, close: bool = False, dxfattribs=None) → MLine`

Add a [MLine](#) entity

Parameters

- **vertices** – MLINE vertices (in [WCS](#))
- **close** – True to add a closed MLINE
- **dfattribs** – additional DXF attributes

`add_spline(fit_points: Iterable[Vertex] = None, degree: int = 3, dxfattribs=None) → Spline`

Add a B-spline ([Spline](#) entity) defined by the given `fit_points` - the control points and knot values are created by the CAD application, therefore it is not predictable how the rendered spline will look like, because for every set of fit points exists an infinite set of B-splines.

If `fit_points` is None, an “empty” spline will be created, all data has to be set by the user.

The SPLINE entity requires DXF R2000.

AutoCAD creates a spline through fit points by a global curve interpolation and an unknown method to estimate the direction of the start- and end tangent.

See also:

- [Tutorial for Spline](#)
- [ezdxf.math.fit_points_to_cad_cv\(\)](#)

Parameters

- **fit_points** – iterable of fit points as (x, y[, z]) in [WCS](#), creates an empty [Spline](#) if None
- **degree** – degree of B-spline, max. degree supported by AutoCAD is 11
- **dfattribs** – additional DXF attributes

`add_cad_spline_control_frame(fit_points: Iterable[Vertex], tangents: Iterable[Vertex] = None, estimate: str = '5-p', dxfattribs=None) → Spline`

Add a [Spline](#) entity passing through the given fit points. This method tries to create the same curve as CAD applications do. To understand the limitations and for more information see function [ezdxf.math.fit_points_to_cad_cv\(\)](#).

Parameters

- **fit_points** – iterable of fit points as (x, y[, z]) in [WCS](#)
- **tangents** – start- and end tangent, default is autodetect
- **estimate** – tangent direction estimation method
- **dfattribs** – additional DXF attributes

`add_spline_control_frame(fit_points: Iterable[Vertex], degree: int = 3, method: str = 'chord', dxfattribs=None) → Spline`

Add a [Spline](#) entity passing through the given `fit_points`, the control points are calculated by a global curve interpolation without start- and end tangent constrains. The new SPLINE entity is defined by control points and not by the fit points, therefore the SPLINE looks always the same, no matter which CAD application renders the SPLINE.

- “uniform”: creates a uniform t vector, from 0 to 1 evenly spaced, see [uniform](#) method
- “distance”, “chord”: creates a t vector with values proportional to the fit point distances, see [chord length](#) method

- “centripetal”, “sqrt_chord”: creates a t vector with values proportional to the fit point sqrt(distances), see [centripetal](#) method
- “arc”: creates a t vector with values proportional to the arc length between fit points.

Use function [`add_cad_spline_control_frame\(\)`](#) to create SPLINE entities from fit points similar to CAD application including start- and end tangent constraints.

Parameters

- **fit_points** – iterable of fit points as (x, y[, z]) in [WCS](#)
- **degree** – degree of B-spline, max. degree supported by AutoCAD is 11
- **method** – calculation method for parameter vector t
- **dxfattribs** – additional DXF attributes

`add_open_spline` (*control_points*: *Iterable[Vertex]*, *degree*: *int* = 3, *knots*: *Iterable[float]* = *None*, *dfattribs*=*None*) → *Spline*

Add an open uniform [Spline](#) defined by *control_points*. (requires DXF R2000)

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

Parameters

- **control_points** – iterable of 3D points in [WCS](#)
- **degree** – degree of B-spline, max. degree supported by AutoCAD is 11
- **knots** – knot values as iterable of floats
- **dfattribs** – additional DXF attributes

`add_rational_spline` (*control_points*: *Iterable[Vertex]*, *weights*: *Sequence[float]*, *degree*: *int* = 3, *knots*: *Iterable[float]* = *None*, *dfattribs*=*None*) → *Spline*

Add an open rational uniform [Spline](#) defined by *control_points*. (requires DXF R2000)

weights has to be an iterable of floats, which defines the influence of the associated control point to the shape of the B-spline, therefore for each control point is one weight value required.

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

Parameters

- **control_points** – iterable of 3D points in [WCS](#)
- **weights** – weight values as iterable of floats
- **degree** – degree of B-spline, max. degree supported by AutoCAD is 11
- **knots** – knot values as iterable of floats
- **dfattribs** – additional DXF attributes

`add_hatch` (*color*: *int* = 7, *dfattribs*=*None*) → *Hatch*

Add a [Hatch](#) entity. (requires DXF R2000)

Parameters

- **color** – fill color as :ref:`ACI`, default is 7 (black/white).
- **dfattribs** – additional DXF attributes

`add_mpolygon` (*color*: *int* = 256, *fill_color*: *int* = *None*, *dfattribs*=*None*) → *MPolygon*

Add a [MPolygon](#) entity. (requires DXF R2000)

The MPOLYGON entity is not a core DXF entity and is not supported by every CAD application or DXF library.

DXF version R2004+ is required to use a fill color different from BYLAYER. For R2000 the fill color is always BYLAYER, set any ACI value to create a filled MPOLYGON entity.

Parameters

- **color** – boundary color as *AutoCAD Color Index (ACI)*, default is BYLAYER.
- **fill_color** – fill color as *AutoCAD Color Index (ACI)*, default is None
- **dxfattribs** – additional DXF attributes

add_mesh (*dxfattribs=None*) → Mesh
Add a *Mesh* entity. (requires DXF R2007)

Parameters **dxfattribs** – additional DXF attributes

add_image (*image_def: ImageDef, insert: Vertex, size_in_units: Tuple[float, float], rotation: float = 0.0, dxfattribs=None*) → Image
Add an *Image* entity, requires a *ImageDef* entity, see *Tutorial for Image and ImageDef*. (requires DXF R2000)

Parameters

- **image_def** – required image definition as *ImageDef*
- **insert** – insertion point as 3D point in *WCS*
- **size_in_units** – size as (x, y) tuple in drawing units
- **rotation** – rotation angle around the extrusion axis, default is the z-axis, in degrees
- **dxfattribs** – additional DXF attributes

add_wipeout (*vertices: Iterable[Vertex], dxfattribs=None*) → Wipeout
Add a *ezdxf.entities.Wipeout* entity, the masking area is defined by WCS *vertices*.

This method creates only a 2D entity in the xy-plane of the layout, the z-axis of the input vertices are ignored.

add_underlay (*underlay_def: UnderlayDefinition, insert: Vertex = (0, 0, 0), scale=(1, 1, 1), rotation: float = 0.0, dxfattribs=None*) → Underlay
Add an *Underlay* entity, requires a *UnderlayDefinition* entity, see *Tutorial for Underlay and UnderlayDefinition*. (requires DXF R2000)

Parameters

- **underlay_def** – required underlay definition as *UnderlayDefinition*
- **insert** – insertion point as 3D point in *WCS*
- **scale** – underlay scaling factor as (x, y, z) tuple or as single value for uniform scaling for x, y and z
- **rotation** – rotation angle around the extrusion axis, default is the z-axis, in degrees
- **dxfattribs** – additional DXF attributes

add_linear_dim (*base: Vertex, p1: Vertex, p2: Vertex, location: Vertex = None, text: str = '<>', angle: float = 0, text_rotation: float = None, dimstyle: str = 'EZDXF', override: Dict[KT, VT] = None, dxfattribs=None*) → DimStyleOverride

Add horizontal, vertical and rotated *Dimension* line. If an *UCS* is used for dimension line rendering, all point definitions in UCS coordinates, translation into *WCS* and *OCS* is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default. See also: *Tutorial for Linear Dimensions*

This method returns a `DimStyleOverride` object - to create the necessary dimension geometry, you have to call `render()` manually, this two step process allows additional processing steps on the `Dimension` entity between creation and rendering.

Note: *Ezdx*f does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Parameters

- **base** – location of dimension line, any point on the dimension line or its extension will do (in UCS)
- **p1** – measurement point 1 and start point of extension line 1 (in UCS)
- **p2** – measurement point 2 and start point of extension line 2 (in UCS)
- **location** – user defined location for text mid point (in UCS)
- **text** – None or “<>” the measurement is drawn as text, ” ” (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **dimstyle** – dimension style name (`DimStyle` table entry), default is “EZDXF”
- **angle** – angle from ucs/wcs x-axis to dimension line in degrees
- **text_rotation** – rotation angle of the dimension text as absolute angle (x-axis=0, y-axis=90) in degrees
- **override** – `DimStyleOverride` attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: `DimStyleOverride`

```
add_multi_point_linear_dim(base: Vertex, points: Iterable[Vertex], angle: float = 0, ucs: UCS
    = None, avoid_double_rendering: bool = True, dimstyle: str =
    'EZDXF', override: Dict[KT, VT] = None, dxfattribs=None, dis-
    card=False) → None
```

Add multiple linear dimensions for iterable *points*. If an `UCS` is used for dimension line rendering, all point definitions in UCS coordinates, translation into `WCS` and `OCS` is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default. See also: [Tutorial for Linear Dimensions](#)

This method sets many design decisions by itself, the necessary geometry will be generated automatically, no required nor possible `render()` call. This method is easy to use but you get what you get.

Note: *Ezdx*f does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Parameters

- **base** – location of dimension line, any point on the dimension line or its extension will do (in UCS)
- **points** – iterable of measurement points (in UCS)
- **angle** – angle from ucs/wcs x-axis to dimension line in degrees (0 = horizontal, 90 = vertical)
- **ucs** – user defined coordinate system

- **avoid_double_rendering** – suppresses the first extension line and the first arrow if possible for continued dimension entities
- **dimstyle** – dimension style name (DimStyle table entry), default is “EZDXF”
- **override** – *DimStyleOverride* attributes
- **dxffattribs** – additional DXF attributes for the DIMENSION entity
- **discard** – discard rendering result for friendly CAD applications like BricsCAD to get a native and likely better rendering result. (does not work with AutoCAD)

add_aligned_dim(*p1*: Vertex, *p2*: Vertex, *distance*: float, *text*: str = '<>', *dimstyle*: str = 'EZDXF', *override*: Dict[KT, VT] = None, *dxffattribs*=None) → DimStyleOverride

Add linear dimension aligned with measurement points *p1* and *p2*. If an *UCS* is used for dimension line rendering, all point definitions in UCS coordinates, translation into *WCS* and *OCS* is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default. See also: [Tutorial for Linear Dimensions](#)

This method returns a *DimStyleOverride* object, to create the necessary dimension geometry, you have to call *DimStyleOverride.render()* manually, this two step process allows additional processing steps on the *Dimension* entity between creation and rendering.

Note: *Ezdx*f does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Parameters

- **p1** – measurement point 1 and start point of extension line 1 (in UCS)
- **p2** – measurement point 2 and start point of extension line 2 (in UCS)
- **distance** – distance of dimension line from measurement points
- **text** – None or “<>” the measurement is drawn as text, ” ” (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **dimstyle** – dimension style name (*DimStyle* table entry), default is “EZDXF”
- **override** – *DimStyleOverride* attributes
- **dxffattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

add_radius_dim(*center*: Vertex, *mpoint*: Vertex = None, *radius*: float = None, *angle*: float = None, *, *location*: Vertex = None, *text*: str = '<>', *dimstyle*: str = 'EZ_RADIUS', *override*: Dict[KT, VT] = None, *dxffattribs*=None) → DimStyleOverride

Add a radius *Dimension* line. The radius dimension line requires a *center* point and a point *mpoint* on the circle or as an alternative a *radius* and a dimension line *angle* in degrees. See also: [Tutorial for Radius Dimensions](#)

If an *UCS* is used for dimension line rendering, all point definitions in UCS coordinates, translation into *WCS* and *OCS* is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default.

This method returns a *DimStyleOverride* object - to create the necessary dimension geometry, you have to call *render()* manually, this two step process allows additional processing steps on the *Dimension* entity between creation and rendering.

Following render types are supported:

- Default text location outside: text aligned with dimension line; dimension style: “EZ_RADIUS”
- Default text location outside horizontal: “EZ_RADIUS” + dimtoh=1
- Default text location inside: text aligned with dimension line; dimension style: “EZ_RADIUS_INSIDE”
- Default text location inside horizontal: “EZ_RADIUS_INSIDE” + dimtih=1
- User defined text location: argument *location* != None, text aligned with dimension line; dimension style: “EZ_RADIUS”
- User defined text location horizontal: argument *location* != None, “EZ_RADIUS” + dimtoh=1 for text outside horizontal, “EZ_RADIUS” + dimtih=1 for text inside horizontal

Placing the dimension text at a user defined *location*, overrides the *mpoint* and the *angle* argument, but requires a given *radius* argument. The *location* argument does not define the exact text location, instead it defines the dimension line starting at *center* and the measurement text midpoint projected on this dimension line going through *location*, if text is aligned to the dimension line. If text is horizontal, *location* is the kink point of the dimension line from radial to horizontal direction.

Note: Ezdxf does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Parameters

- **center** – center point of the circle (in UCS)
- **mpoint** – measurement point on the circle, overrides *angle* and *radius* (in UCS)
- **radius** – radius in drawing units, requires argument *angle*
- **angle** – specify angle of dimension line in degrees, requires argument *radius*
- **location** – user defined dimension text location, overrides *mpoint* and *angle*, but requires *radius* (in UCS)
- **text** – None or “<>” the measurement is drawn as text, ” ” (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **dimstyle** – dimension style name (*DimStyle* table entry), default is “EZ_RADIUS”
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

add_radius_dim_2p(*center*: Vertex, *mpoint*: Vertex, *, *text*: str = '<>', *dimstyle*: str = 'EZ_RADIUS', *override*: Dict[KT, VT] = None, *dxfattribs*=None) → DimStyleOverride

Shortcut method to create a radius dimension by center point, measurement point on the circle and the measurement text at the default location defined by the associated *dimstyle*, for further information see general method [add_radius_dim\(\)](#).

- *dimstyle* “EZ_RADIUS”: places the dimension text outside
- *dimstyle* “EZ_RADIUS_INSIDE”: places the dimension text inside

Parameters

- **center** – center point of the circle (in UCS)

- **mpoint** – measurement point on the circle (in UCS)
- **text** – `None` or “`<>`” the measurement is drawn as text, “ ” (a single space) suppresses the dimension text, everything else `text` is drawn as dimension text
- **dimstyle** – dimension style name (`DimStyle` table entry), default is “EZ_RADIUS”
- **override** – `DimStyleOverride` attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: `DimStyleOverride`

add_radius_dim_cra(center: `Vertex`, radius: `float`, angle: `float`, *, text: `str` = ‘`<>`’, dimstyle: `str` = ‘EZ_RADIUS’, override: `Dict[KT, VT] = None`, dxfattribs=`None`) → `DimStyleOverride`

Shortcut method to create a radius dimension by (c)enter point, (r)adius and (a)ngle, the measurement text is placed at the default location defined by the associated `dimstyle`, for further information see general method `add_radius_dim()`.

- `dimstyle` “EZ_RADIUS”: places the dimension text outside
- `dimstyle` “EZ_RADIUS_INSIDE”: places the dimension text inside

Parameters

- **center** – center point of the circle (in UCS)
- **radius** – radius in drawing units
- **angle** – angle of dimension line in degrees
- **text** – `None` or “`<>`” the measurement is drawn as text, “ ” (a single space) suppresses the dimension text, everything else `text` is drawn as dimension text
- **dimstyle** – dimension style name (`DimStyle` table entry), default is “EZ_RADIUS”
- **override** – `DimStyleOverride` attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: `DimStyleOverride`

add_diameter_dim(center: `Vertex`, mpoint: `Vertex` = `None`, radius: `float` = `None`, angle: `float` = `None`, *, location: `Vertex` = `None`, text: `str` = ‘`<>`’, dimstyle: `str` = ‘EZ_RADIUS’, override: `Dict[KT, VT] = None`, dxfattribs=`None`) → `DimStyleOverride`

Add a diameter `Dimension` line. The diameter dimension line requires a `center` point and a point `mpoint` on the circle or as an alternative a `radius` and a dimension line `angle` in degrees.

If an `UCS` is used for dimension line rendering, all point definitions in UCS coordinates, translation into `WCS` and `OCS` is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default.

This method returns a `DimStyleOverride` object - to create the necessary dimension geometry, you have to call `render()` manually, this two step process allows additional processing steps on the `Dimension` entity between creation and rendering.

Note: `Ezdx` does not consider all `DIMSTYLE` variables, so the rendering results are different from CAD applications.

Parameters

- **center** – specifies the center of the circle (in UCS)
- **mpoint** – specifies the measurement point on the circle (in UCS)
- **radius** – specify radius, requires argument *angle*, overrides *p1* argument
- **angle** – specify angle of dimension line in degrees, requires argument *radius*, overrides *p1* argument
- **location** – user defined location for text mid point (in UCS)
- **text** – None or "<>" the measurement is drawn as text, " " (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **dimstyle** – dimension style name (*DimStyle* table entry), default is “EZ_RADIUS”
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

add_diameter_dim_2p(*p1*: Vertex, *p2*: Vertex, *text*: str = '<>', *dimstyle*: str = 'EZ_RADIUS', *override*: Dict[KT, VT] = None, *dxfattribs*=None) → DimStyleOverride

Shortcut method to create a diameter dimension by two points on the circle and the measurement text at the default location defined by the associated *dimstyle*, for further information see general method [add_diameter_dim\(\)](#). Center point of the virtual circle is the mid point between *p1* and *p2*.

- dimstyle “EZ_RADIUS”: places the dimension text outside
- dimstyle “EZ_RADIUS_INSIDE”: places the dimension text inside

Parameters

- **p1** – first point of the circle (in UCS)
- **p2** – second point on the opposite side of the center point of the circle (in UCS)
- **text** – None or “<>” the measurement is drawn as text, “ ” (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **dimstyle** – dimension style name (*DimStyle* table entry), default is “EZ_RADIUS”
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

add_angular_dim_2l(*base*: Vertex, *line1*: Tuple[Vertex, Vertex], *line2*: Tuple[Vertex, Vertex], *, *location*: Vertex = None, *text*: str = '<>', *text_rotation*: float = None, *dimstyle*: str = 'EZ_CURVED', *override*: Dict[KT, VT] = None, *dxfattribs*=None) → DimStyleOverride

Add angular *Dimension* from 2 lines. The measurement is always done from *line1* to *line2* in counter clockwise orientation. This does not always match the result in CAD applications!

If an *UCS* is used for angular dimension rendering, all point definitions in UCS coordinates, translation into *WCS* and *OCS* is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default.

This method returns a *DimStyleOverride* object - to create the necessary dimension geometry, you have to call [render\(\)](#) manually, this two step process allows additional processing steps on the *Dimension* entity between creation and rendering.

Note: *Ezdx*f does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Parameters

- **base** – location of dimension line, any point on the dimension line or its extension is valid (in UCS)
- **line1** – specifies start leg of the angle (start point, end point) and determines extension line 1 (in UCS)
- **line2** – specifies end leg of the angle (start point, end point) and determines extension line 2 (in UCS)
- **location** – user defined location for text mid point (in UCS)
- **text** – None or “<>” the measurement is drawn as text, “ ” (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **text_rotation** – rotation angle of the dimension text as absolute angle (x-axis=0, y-axis=90) in degrees
- **dimstyle** – dimension style name (*DimStyle* table entry), default is “EZ_CURVED”
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

New in version v0.18.

add_angular_dim_3p(*base*: Vertex, *center*: Vertex, *p1*: Vertex, *p2*: Vertex, *, *location*: Vertex = None, *text*: str = '<>', *text_rotation*: float = None, *dimstyle*: str = 'EZ_CURVED', *override*: Dict[KT, VT] = None, *dxfattribs*=None) → DimStyleOverride

Add angular *Dimension* from 3 points (center, p1, p2). The measurement is always done from *p1* to *p2* in counter clockwise orientation. This does not always match the result in CAD applications!

If an *UCS* is used for angular dimension rendering, all point definitions in UCS coordinates, translation into *WCS* and *OCS* is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default.

This method returns a *DimStyleOverride* object - to create the necessary dimension geometry, you have to call *render()* manually, this two step process allows additional processing steps on the *Dimension* entity between creation and rendering.

Note: *Ezdx*f does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Parameters

- **base** – location of dimension line, any point on the dimension line or its extension is valid (in UCS)
- **center** – specifies the vertex of the angle
- **p1** – specifies start leg of the angle (center -> p1) and end point of extension line 1 (in UCS)

- **p2** – specifies end leg of the angle (center -> p2) and end point of extension line 2 (in UCS)
- **location** – user defined location for text mid point (in UCS)
- **text** – None or “<>” the measurement is drawn as text, ” ” (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **text_rotation** – rotation angle of the dimension text as absolute angle (x-axis=0, y-axis=90) in degrees
- **dimstyle** – dimension style name (*DimStyle* table entry), default is “EZ_CURVED”
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

New in version v0.18.

add_angular_dim_cra(*center*: Vertex, *radius*: float, *start_angle*: float, *end_angle*: float, *distance*: float, *, *location*: Vertex = None, *text*: str = '<>', *text_rotation*: float = None, *dimstyle*: str = 'EZ_CURVED', *override*: Dict[KT, VT] = None, *dxfattribs*=None) → DimStyleOverride

Shortcut method to create an angular dimension by (c)enter point, (r)adius and start- and end (a)ngles, the measurement text is placed at the default location defined by the associated *dimstyle*. The measurement is always done from *start_angle* to *end_angle* in counter clockwise orientation. This does not always match the result in CAD applications! For further information see the more generic factory method *add_angular_dim_3p()*.

Parameters

- **center** – center point of the angle (in UCS)
- **radius** – the distance from *center* to the start of the extension lines in drawing units
- **start_angle** – start angle in degrees (in UCS)
- **end_angle** – end angle in degrees (in UCS)
- **distance** – distance from start of the extension lines to the dimension line in drawing units
- **location** – user defined location for text mid point (in UCS)
- **text** – None or “<>” the measurement is drawn as text, ” ” (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **text_rotation** – rotation angle of the dimension text as absolute angle (x-axis=0, y-axis=90) in degrees
- **dimstyle** – dimension style name (*DimStyle* table entry), default is “EZ_CURVED”
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

New in version v0.18.

```
add_angular_dim_arc(arc: eздxf.math.arc.ConstructionArc, distance: float, *, location: Vertex
= None, text: str = '<>', text_rotation: float = None, dimstyle: str =
'EZ_CURVED', override: Dict[KT, VT] = None, dxfsattribs=None) →
DimStyleOverride
```

Shortcut method to create an angular dimension from a `ConstructionArc`. This construction tool can be created from ARC entities and the tool itself provides various construction class methods. The measurement text is placed at the default location defined by the associated `dimstyle`. The measurement is always done from `start_angle` to `end_angle` of the arc in counter clockwise orientation. This does not always match the result in CAD applications! For further information see the more generic factory method `add_angular_dim_3p()`.

Parameters

- **arc** – `ConstructionArc`
- **distance** – distance from start of the extension lines to the dimension line in drawing units
- **location** – user defined location for text mid point (in UCS)
- **text** – `None` or “<>” the measurement is drawn as text, “ ” (a single space) suppresses the dimension text, everything else `text` is drawn as dimension text
- **text_rotation** – rotation angle of the dimension text as absolute angle (x-axis=0, y-axis=90) in degrees
- **dimstyle** – dimension style name (`DimStyle` table entry), default is “EZ_CURVED”
- **override** – `DimStyleOverride` attributes
- **dxfsattribs** – additional DXF attributes for the DIMENSION entity

Returns: `DimStyleOverride`

New in version v0.18.

```
add_arc_dim_3p(base: Vertex, center: Vertex, p1: Vertex, p2: Vertex, *, location: Vertex = None, text:
str = '<>', text_rotation: float = None, dimstyle: str = 'EZ_CURVED', override:
Dict[KT, VT] = None, dxfsattribs=None) → DimStyleOverride
```

Add `ArcDimension` from 3 points (center, p1, p2). Point `p1` defines the radius and the start angle of the arc, point `p2` only defines the end angle of the arc.

If an `UCS` is used for arc dimension rendering, all point definitions in UCS coordinates, translation into `WCS` and `OCS` is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default.

This method returns a `DimStyleOverride` object - to create the necessary dimension geometry, you have to call `render()` manually, this two step process allows additional processing steps on the `ArcDimension` entity between creation and rendering.

Note: *Ezdx*f does not render the arc dimension like CAD applications and does not consider all DIMSTYLE variables, so the rendering results are **very** different from CAD applications.

Parameters

- **base** – location of dimension line, any point on the dimension line or its extension is valid (in UCS)
- **center** – specifies the vertex of the angle
- **p1** – specifies the radius (center -> p1) and the star angle of the arc, this is also the start point for the 1st extension line (in UCS)

- **p2** – specifies the end angle of the arc. The start 2nd extension line is defined by this angle and the radius defined by p1 (in UCS)
- **location** – user defined location for text mid point (in UCS)
- **text** – None or “<>” the measurement is drawn as text, ” ” (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **text_rotation** – rotation angle of the dimension text as absolute angle (x-axis=0, y-axis=90) in degrees
- **dimstyle** – dimension style name (*DimStyle* table entry), default is “EZ_CURVED”
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

New in version v0.18.

add_arc_dim_cra (*center*: Vertex, *radius*: float, *start_angle*: float, *end_angle*: float, *distance*: float, *, *location*: Vertex = None, *text*: str = '<>', *text_rotation*: float = None, *dimstyle*: str = 'EZ_CURVED', *override*: Dict[KT, VT] = None, *dxfattribs*=None)
→ *DimStyleOverride*

Shortcut method to create an arc dimension by (c)enter point, (r)adius and start- and end (a)ngles, the measurement text is placed at the default location defined by the associated *dimstyle*.

Note: *Ezdx*f does not render the arc dimension like CAD applications and does not consider all DIMSTYLE variables, so the rendering results are **very** different from CAD applications.

Parameters

- **center** – center point of the angle (in UCS)
- **radius** – the distance from *center* to the start of the extension lines in drawing units
- **start_angle** – start angle in degrees (in UCS)
- **end_angle** – end angle in degrees (in UCS)
- **distance** – distance from start of the extension lines to the dimension line in drawing units
- **location** – user defined location for text mid point (in UCS)
- **text** – None or “<>” the measurement is drawn as text, ” ” (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **text_rotation** – rotation angle of the dimension text as absolute angle (x-axis=0, y-axis=90) in degrees
- **dimstyle** – dimension style name (*DimStyle* table entry), default is “EZ_CURVED”
- **override** – *DimStyleOverride* attributes
- **dxfattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

New in version v0.18.

```
add_arc_dim_arc(arc: ezdxf.math.arc.ConstructionArc, distance: float, *, location: Vertex = None,
text: str = '<>', text_rotation: float = None, dimstyle: str = 'EZ_CURVED',
override: Dict[KT, VT] = None, dxftattribs=None) → DimStyleOverride
```

Shortcut method to create an arc dimension from a `ConstructionArc`. This construction tool can be created from ARC entities and the tool itself provides various construction class methods. The measurement text is placed at the default location defined by the associated `dimstyle`.

Note: *Ezdx*f does not render the arc dimension like CAD applications and does not consider all DIMSTYLE variables, so the rendering results are **very** different from CAD applications.

Parameters

- **arc** – `ConstructionArc`
- **distance** – distance from start of the extension lines to the dimension line in drawing units
- **location** – user defined location for text mid point (in UCS)
- **text** – None or “<>” the measurement is drawn as text, ” ” (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **text_rotation** – rotation angle of the dimension text as absolute angle (x-axis=0, y-axis=90) in degrees
- **dimstyle** – dimension style name (`DimStyle` table entry), default is “EZ_CURVED”
- **override** – `DimStyleOverride` attributes
- **dxftattribs** – additional DXF attributes for the DIMENSION entity

Returns: `DimStyleOverride`

New in version v0.18.

```
add_ordinate_dim(feature_location: Vertex, offset: Vertex, dtype: int, *, origin: Vertex =
Vec3(0.0, 0.0, 0.0), rotation: float = 0.0, text: str = '<>', dimstyle:
str = 'EZDXF', override: Dict[KT, VT] = None, dxftattribs=None) →
ezdxf.entities.dimstyleoverride.DimStyleOverride
```

Add an ordinate type `Dimension` line. The feature location is defined in the global coordinate system, which is set as render UCS, which is the `WCS` by default.

If an `UCS` is used for dimension line rendering, all point definitions in UCS coordinates, translation into `WCS` and `OCS` is done by the rendering function. Extrusion vector is defined by UCS or (0, 0, 1) by default.

This method returns a `DimStyleOverride` object - to create the necessary dimension geometry, you have to call `render()` manually, this two step process allows additional processing steps on the `Dimension` entity between creation and rendering.

Note: *Ezdx*f does not consider all DIMSTYLE variables, so the rendering results are different from CAD applications.

Parameters

- **feature_location** – feature location in the global coordinate system (UCS)
- **offset** – offset vector of leader end point from the feature location in the local coordinate system

- **dtype** – 1 = x-type, 0 = y-type
- **origin** – specifies the origin (0, 0) of the local coordinate system in UCS
- **rotation** – rotation angle of the local coordinate system in degrees
- **text** – None or “<>” the measurement is drawn as text, ” ” (a single space) suppresses the dimension text, everything else *text* is drawn as dimension text
- **dimstyle** – dimension style name (*DimStyle* table entry), default is “EZDXF”
- **override** – *DimStyleOverride* attributes
- **dxftattribs** – additional DXF attributes for the DIMENSION entity

Returns: *DimStyleOverride*

New in version v0.18.

```
add_ordinate_x_dim(feature_location: Vertex, offset: Vertex, *, origin: Vertex = Vec3(0.0, 0.0, 0.0), rotation: float = 0.0, text: str = '<>', dimstyle: str = 'EZDXF', override: Dict[KT, VT] = None, dxftattribs=None) → ezdxf.entities.dimstyleoverride.DimStyleOverride
```

Shortcut to add a x-type feature ordinate DIMENSION, for more information see [add_ordinate_dim\(\)](#).

New in version v0.18.

```
add_ordinate_y_dim(feature_location: Vertex, offset: Vertex, *, origin: Vertex = Vec3(0.0, 0.0, 0.0), rotation: float = 0.0, text: str = '<>', dimstyle: str = 'EZDXF', override: Dict[KT, VT] = None, dxftattribs=None) → ezdxf.entities.dimstyleoverride.DimStyleOverride
```

Shortcut to add a y-type feature ordinate DIMENSION, for more information see [add_ordinate_dim\(\)](#).

New in version v0.18.

```
add_leader(vertices: Iterable[Vertex], dimstyle: str = 'EZDXF', override: Dict[KT, VT] = None, dxftattribs=None) → Leader
```

The *Leader* entity represents an arrow, made up of one or more vertices (or spline fit points) and an arrowhead. The label or other content to which the *Leader* is attached is stored as a separate entity, and is not part of the *Leader* itself. (requires DXF R2000)

Leader shares its styling infrastructure with *Dimension*.

By default a *Leader* without any annotation is created. For creating more fancy leaders and annotations see documentation provided by Autodesk or [Demystifying DXF: LEADER and MULTILEADER implementation notes](#).

Parameters

- **vertices** – leader vertices (in *WCS*)
- **dimstyle** – dimension style name (*DimStyle* table entry), default is “EZDXF”
- **override** – override *DimStyleOverride* attributes
- **dxftattribs** – additional DXF attributes

```
add_multileader_mtext(style: str = 'Standard', dxftattribs=None) → MultiLeaderMTextBuilder
```

Add a *MultiLeader* entity but returns a *MultiLeaderMTextBuilder*.

New in version 0.18.

add_multileader_block (*style: str = 'Standard'*, *dxfattribs=None*) → MultiLeaderBlockBuilder
Add a [MultiLeader](#) entity but returns a [MultiLeaderBlockBuilder](#).

New in version 0.18.

add_body (*acis_data: Iterable[str] = None*, *dxfattribs=None*) → Body
Add a [Body](#) entity. (requires DXF R2000-R2010)

The ACIS data has to be provided as an iterable of strings with no line endings and only the SAT (text) format is supported, DXF R2013+ uses the SAB (binary) format which is not supported, and *ezdxf* has no functionality to create ACIS data.

Parameters

- **acis_data** – ACIS data as iterable of text lines as strings, no interpretation by *ezdxf* possible
- **dfattribs** – additional DXF attributes

add_region (*acis_data: Iterable[str] = None*, *dxfattribs=None*) → Region
Add a [Region](#) entity. (requires DXF R2000-R2010)

The ACIS data has to be provided as an iterable of strings with no line endings and only the SAT (text) format is supported, DXF R2013+ uses the SAB (binary) format which is not supported, and *ezdxf* has no functionality to create ACIS data.

Parameters

- **acis_data** – ACIS data as iterable of text lines as strings, no interpretation by *ezdxf* possible
- **dfattribs** – additional DXF attributes

add_3dsolid (*acis_data: Iterable[str] = None*, *dxfattribs=None*) → Solid3d
Add a 3DSOLID entity ([Solid3d](#)). (requires DXF R2000-R2010)

The ACIS data has to be provided as an iterable of strings with no line endings and only the SAT (text) format is supported, DXF R2013+ uses the SAB (binary) format which is not supported, and *ezdxf* has no functionality to create ACIS data.

Parameters

- **acis_data** – ACIS data as iterable of text lines as strings, no interpretation by *ezdxf* possible
- **dfattribs** – additional DXF attributes

add_surface (*acis_data: Iterable[str] = None*, *dxfattribs=None*) → Surface
Add a [Surface](#) entity. (requires DXF R2000-R2010)

The ACIS data has to be provided as an iterable of strings with no line endings and only the SAT (text) format is supported, DXF R2013+ uses the SAB (binary) format which is not supported, and *ezdxf* has no functionality to create ACIS data.

Parameters

- **acis_data** – ACIS data as iterable of text lines as strings, no interpretation by *ezdxf* possible
- **dfattribs** – additional DXF attributes

add_extruded_surface (*acis_data: Iterable[str] = None*, *dxfattribs=None*) → ExtrudedSurface
Add a [ExtrudedSurface](#) entity. (requires DXF R2000-R2010)

The ACIS data has to be provided as an iterable of strings with no line endings and only the SAT (text) format is supported, DXF R2013+ uses the SAB (binary) format which is not supported, and *ezdxf* has no functionality to create ACIS data.

Parameters

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

add_lofted_surface (*acis_data*: *Iterable[str]* = *None*, *dxfattribs*=*None*) → LoftedSurface

Add a *LoftedSurface* entity. (requires DXF R2007-R2010)

The ACIS data has to be provided as an iterable of strings with no line endings and only the SAT (text) format is supported, DXF R2013+ uses the SAB (binary) format which is not supported, and *ezdxf* has no functionality to create ACIS data.

Parameters

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

add_revolved_surface (*acis_data*: *Iterable[str]* = *None*, *dxfattribs*=*None*) → RevolvedSurface

Add a *RevolvedSurface* entity. (requires DXF R2007-R2010)

The ACIS data has to be provided as an iterable of strings with no line endings and only the SAT (text) format is supported, DXF R2013+ uses the SAB (binary) format which is not supported, and *ezdxf* has no functionality to create ACIS data.

Parameters

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

add_swept_surface (*acis_data*: *Iterable[str]* = *None*, *dxfattribs*=*None*) → SweptSurface

Add a *SweptSurface* entity. (requires DXF R2007-R2010)

The ACIS data has to be provided as an iterable of strings with no line endings and only the SAT (text) format is supported, DXF R2013+ uses the SAB (binary) format which is not supported, and *ezdxf* has no functionality to create ACIS data.

Parameters

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

Layout

class eздxf.layouts.**Layout**

Layout is a subclass of *BaseLayout* and common base class of *Modelspace* and *Paperspace*.

name

Layout name as shown in tabs of *CAD* applications.

dxfs

Returns the DXF name space attribute of the associated `DXFLayout` object.

This enables direct access to the underlying LAYOUT entity, e.g. `Layout.dxf.layout_flags`

__contains__(entity: Union[DXFGraphic, str]) → bool

Returns True if `entity` is stored in this layout.

Parameters `entity` – DXFGraphic object or handle as hex string

reset_extents(extmin=(1e+20, 1e+20, 1e+20), extmax=(-1e+20, -1e+20, -1e+20)) → None

Reset `extents` to given values or the AutoCAD default values.

“Drawing extents are the bounds of the area occupied by objects.” (Quote Autodesk Knowledge Network)

Parameters

- `extmin` – minimum extents or (1e20, 1e20, 1e20) as default value
- `extmax` – maximum extents or (-1e20, -1e20, -1e20) as default value

reset_limits(limmin=None, limmax=None) → None

Reset `limits` to given values or the AutoCAD default values.

“Sets an invisible rectangular boundary in the drawing area that can limit the grid display and limit clicking or entering point locations.” (Quote Autodesk Knowledge Network)

The `Paperspace` class has an additional method `reset_paper_limits()` to deduce the default limits from the paper size settings.

Parameters

- `extmin` – minimum extents or (0, 0) as default
- `extmax` – maximum extents or (paper width, paper height) as default value

set_plot_type(value: int = 5) → None

0	last screen display
1	drawing extents
2	drawing limits
3	view specific (defined by <code>Layout.dxf.plot_view_name</code>)
4	window specific (defined by <code>Layout.set_plot_window_limits()</code>)
5	layout information (default)

Parameters `value` – plot type

Raises `DXFValueError` – for `value` out of range

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

Set plot style file of type `.ctb`.

Parameters

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

set_plot_window(lower_left: Tuple[float, float] = (0, 0), upper_right: Tuple[float, float] = (0, 0))

→ `None`

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

Parameters

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

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

If the header variable \$SORTENTS Regen flag (bit-code value 16) is set, AutoCAD regenerates entities in ascending handles order.

To change redraw order associate a different sort handle to entities, this redefines the order in which the entities are regenerated. *handles* can be a dict of entity_handle and sort_handle as (k, v) pairs, or an iterable of (entity_handle, sort_handle) tuples.

The sort_handle doesn't have to be unique, some or all entities can share the same sort handle and a sort handle can be an existing handle.

The “0” handle can be used, but this sort_handle will be drawn as latest (on top of all other entities) and not as first as expected.

Parameters **handles** – iterable or dict of handle associations; an iterable of 2-tuples (entity_handle, sort_handle) or a dict (k, v) association as (entity_handle, sort_handle)

get_redraw_order () → Iterable[Tuple[str, str]]

Returns iterable for all existing table entries as (entity_handle, sort_handle) pairs, see also [set_redraw_order](#) ().

plot_viewport_borders (*state*: bool = True) → None

show_plot_styles (*state*: bool = True) → None

plot_centered (*state*: bool = True) → None

plot_hidden (*state*: bool = True) → None

use_standard_scale (*state*: bool = True) → None

use_plot_styles (*state*: bool = True) → None

scale_lineweights (*state*: bool = True) → None

print_lineweights (*state*: bool = True) → None

draw_viewports_first (*state*: bool = True) → None

model_type (*state*: bool = True) → None

update_paper (*state*: bool = True) → None

zoom_to_paper_on_update (*state*: bool = True) → None

plot_flags_initializing (*state*: bool = True) → None

prev_plot_init (*state*: bool = True) → None

set_plot_flags (*flag, state*: bool = True) → None

Modelspace

class eздxf.layouts.**Modelspace**

Modelspace is a subclass of [Layout](#).

The modelspace contains the “real” world representation of the drawing subjects in real world units.

name

Name of modelspace is fixed as “Model”.

new_geodata (*dxfattribs: dict = None*) → GeoData

Creates a new GeoData entity and replaces existing ones. The GEODATA entity resides in the OBJECTS section and not in the modelspace, it is linked to the modelspace by an ExtensionDict located in BLOCK_RECORD of the modelspace.

The GEODATA entity requires DXF R2010. The DXF reference does not document if other layouts than the modelspace supports geo referencing, so I assume getting/setting geo data may only make sense for the modelspace.

Parameters **dxfattribs** – DXF attributes for *GeoData* entity

get_geodata () → Optional[GeoData]

Returns the *GeoData* entity associated to the modelspace or None.

Paperspace

class ezdxf.layouts.Paperspace

Paperspace is a subclass of *Layout*.

Paperspace layouts are used to create different drawing sheets of the modelspace subjects for printing or PDF export.

name

Layout name as shown in tabs of *CAD* applications.

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

Setup plot settings and paper size and reset viewports. All parameters in given *units* (mm or inch).

Reset paper limits, extents and viewports.

Parameters

- **size** – paper size as (width, height) tuple
- **margins** – (top, right, bottom, left) hint: clockwise
- **units** – “mm” or “inch”
- **offset** – plot origin offset is 2D point
- **rotation** – see table Rotation
- **scale** – integer in range [0, 32] defines a standard scale type or as tuple(numerator, denominator) e.g. (1, 50) for scale 1:50
- **name** – paper name prefix “{name}_{width}_x_{height}_{unit}”
- **device** – device .pc3 configuration file or system printer name

int	Rotation
0	no rotation
1	90 degrees counter-clockwise
2	upside-down
3	90 degrees clockwise

viewports () → List[Viewport]

Get all VIEWPORT entities defined in this paperspace layout. Returns a list of *Viewport* objects, sorted by id, the first entity is always the main viewport with an id of 1.

main_viewport() → Optional[Viewport]

Returns the main viewport of this paper space layout, or `None` if no main viewport exist.

add_viewport(*center*: Vertex, *size*: Tuple[float, float], *view_center_point*: Vertex, *view_height*: float, *dxfattribs*: dict = None) → Viewport

Add a new `Viewport` entity.

reset_viewports() → None

Delete all existing viewports, and create a new main viewport.

reset_main_viewport(*center*: Vertex = None, *size*: Vertex = None) → Viewport

Reset the main viewport of this paper space layout to the given values, or reset them to the default values, deduced from the paper settings. Creates a new main viewport if none exist.

Ezdx does not create a main viewport by default, because CAD applications don't require one.

Parameters

- **center** – center of the viewport in paper space units

- **size** – viewport size as (width, height) tuple in paper space units

reset_paper_limits() → None

Set paper limits to default values, all values in paperspace units but without plot scale (?).

get_paper_limits() → Tuple[Vec2, Vec2]

Returns paper limits in plot paper units, relative to the plot origin.

plot origin = lower left corner of printable area + plot origin offset

Returns tuple (Vec2(x1, y1), Vec2(x2, y2)), lower left corner is (x1, y1), upper right corner is (x2, y2).

BlockLayout

class ezdxf.layouts.BlockLayout

`BlockLayout` is a subclass of `BaseLayout`.

Block layouts are reusable sets of graphical entities, which can be referenced by multiple `Insert` entities. Each reference can be placed, scaled and rotated individually and can have its own set of DXF `Attrib` entities attached.

__contains__(*entity*) → bool

Returns True if block contains *entity*.

Parameters **entity** – DXFGraphic object or handle as hex string

attdefs() → Iterable[ezdxf.entities.attrib.AttDef]

Returns iterable of all Attdef entities.

has_attdef(*tag*: str) → bool

Returns True if an Attdef for *tag* exist.

get_attdef(*tag*: str) → Optional[ezdxf.entities.dxfgfx.DXFGraphic]

Returns attached Attdef entity by *tag* name.

get_attdef_text(*tag*: str, *default*: str = "") → str

Returns text content for Attdef *tag* as string or returns *default* if no Attdef for *tag* exist.

Parameters

- **tag** – name of tag

- **default** – default value if *tag* not exist

Groups

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

Group entities have to be in modelspace or any paperspace layout but not in a block definition!

DXFGroup

class `ezdxf.entities.dxfgroups.DXFGroup`

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

`dxf.description`

group description (string)

`dxf.unnamed`

1 for unnamed, 0 for named group (int)

`dxf.selectable`

1 for selectable, 0 for not selectable group (int)

`__iter__()` → `Iterator[ezdxf.entities.dxfentity.DXFEntity]`

Iterate over all DXF entities in `DXFGroup` as instances of `DXFGraphic` or inherited (LINE, CIRCLE, ...).

`__len__()` → int

Returns the count of DXF entities in `DXFGroup`.

`__getitem__(item)`

Returns entities by standard Python indexing and slicing.

`__contains__(item: Union[str, ezdxf.entities.dxfentity.DXFEntity])` → bool

Returns True if item is in `DXFGroup`. `item` has to be a handle string or an object of type `DXFEntity` or inherited.

`handles()` → Iterable[str]

Iterable of handles of all DXF entities in `DXFGroup`.

`edit_data()` → List[`ezdxf.entities.dxfentity.DXFEntity`]

Context manager which yields all the group entities as standard Python list:

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

`set_data(entities: Iterable[ezdxf.entities.dxfentity.DXFEntity])` → None

Set `entities` as new group content, entities should be an iterable `DXFGraphic` or inherited (LINE, CIRCLE, ...). Raises `DXFValueError` if not all entities be on the same layout (modelspace or any paperspace layout but not block)

`extend(entities: Iterable[ezdxf.entities.dxfentity.DXFEntity])` → None

Add `entities` to `DXFGroup` without immediate verification!

Validation at DXF export may raise a `DXFStructureError`!

clear() → None

Remove all entities from *DXFGroup*, does not delete any drawing entities referenced by this group.

audit(auditor: Auditor) → None

Remove invalid entities from *DXFGroup*.

Invalid entities are:

- deleted entities
- all entities which do not reside in model- or paper space
- all entities if they do not reside in the same layout

GroupCollection

Each *Drawing* has one group table, which is accessible by the attribute *groups*.

class eздxf.entities.dxfgroups.GroupCollection

Manages all *DXFGroup* objects of a *Drawing*.

__len__() → int

Returns the count of DXF groups.

__iter__()

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

__contains__(name: str) → bool

Returns True if a group *name* exist.

get(name: str) → DXFGroup

Returns the group *name*. Raises *DXFKeyError* if group *name* does not exist.

groups() → DXFGroup

Iterable of all existing groups.

new(name: str=None, description: str='', selectable: bool=True) → DXFGroup

Creates a new group. If *name* is None an unnamed group is created, which has an automatically generated name like “*Annnn”. Group names are case insensitive.

Parameters

- **name** – group name as string
- **description** – group description as string
- **selectable** – group is selectable if True

delete(group: Union[DXFGroup, str]) → None

Delete *group*, *group* can be an object of type *DXFGroup* or a group name as string.

clear()

Delete all groups.

audit(auditor: Auditor) → None

Removes empty groups and invalid handles from all groups.

DXF Entities

All DXF entities can only reside in the *BaseLayout* and inherited classes like *Modelspace*, *Paperspace* and *BlockLayout*.

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

DXF Entity Base Class

Common base class for all DXF entities and objects.

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

class ezdxf.entities.**DXFEntity**

dxfs

The DXF attributes namespace:

```
# set attribute value
entity.dxf.layer = 'MyLayer'

# get attribute value
linetype = entity.dxf.linetype

# delete attribute
del entity.dxf.linetype
```

dxfs.handle

DXF *handle* is a unique identifier as plain hex string like F000. (feature for experts)

dxfs.owner

Handle to *owner* as plain hex string like F000. (feature for experts)

doc

Get the associated *Drawing* instance.

dxftype() → str

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

__str__() → str

Returns a simple string representation.

__repr__() → str

Returns a simple string representation including the class.

has_dxf_attrib(key: str) → bool

Returns True if DXF attribute *key* really exist.

Raises DXFAttributeError if *key* is not an supported DXF attribute.

is_supported_dxf_attrib(key: str) → bool

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

get_dxf_attrib(key: str, default: Any = None) → Any

Get DXF attribute *key*, returns *default* if key doesn't exist, or raise DXFValueError if *default* is DXFValueError and no DXF default value is defined:

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

Raises DXFAttributeError if *key* is not an supported DXF attribute.

set_dxf_attrib(*key: str, value: Any*) → None

Set new *value* for DXF attribute *key*:

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

Raises DXFAttributeError if *key* is not an supported DXF attribute.

del_dxf_attrib(*key: str*) → None

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

Raises DXFAttributeError if *key* is not an supported DXF attribute.

dxfattribs(*drop: Set[str] = None*) → Dict[KT, VT]

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

update_dxf_attribs(*dxfattribs: Dict[KT, VT]*) → None

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

set_flag_state(*flag: int, state: bool = True, name: str = 'flags'*) → None

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

get_flag_state(*flag: int, name: str = 'flags'*) → bool

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

has_extension_dict

Returns True if entity has an attached *ExtensionDict* instance.

get_extension_dict() → ExtensionDict

Returns the existing *ExtensionDict* instance.

Raises AttributeError – extension dict does not exist

new_extension_dict() → ExtensionDict

Create a new *ExtensionDict* instance .

discard_extension_dict() → None

Delete *ExtensionDict* instance .

has_app_data(*appid: str*) → bool

Returns True if application defined data for *appid* exist.

get_app_data(*appid: str*) → Tags

Returns application defined data for *appid*.

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

Raises DXFValueError – no data for *appid* found

set_app_data(*appid: str, tags: Iterable*)

Set application defined data for *appid* as iterable of tags.

Parameters

- **appid** – application name as defined in the APPID table.

- **tags** – iterable of (code, value) tuples or *DXFTag*

discard_app_data(*appid: str*)

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

has_xdata (*appid*: str) → bool
 Returns True if extended data for *appid* exist.

get_xdata (*appid*: str) → Tags
 Returns extended data for *appid*.

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

Raises DXFValueError – no extended data for *appid* found

set_xdata (*appid*: str, *tags*: Iterable)
 Set extended data for *appid* as iterable of tags.

Parameters

- **appid** – application name as defined in the APPID table.
- **tags** – iterable of (code, value) tuples or *DXFTag*

discard_xdata (*appid*: str) → None
 Discard extended data for *appid*. Does not raise an exception if no extended data for *appid* exist.

has_xdata_list (*appid*: str, *name*: str) → bool
 Returns True if a tag list *name* for extended data *appid* exist.

get_xdata_list (*appid*: str, *name*: str) → Tags
 Returns tag list *name* for extended data *appid*.

Parameters

- **appid** – application name as defined in the APPID table.
- **name** – extended data list name

Raises DXFValueError – no extended data for *appid* found or no data list *name* not found

set_xdata_list (*appid*: str, *name*: str, *tags*: Iterable)
 Set tag list *name* for extended data *appid* as iterable of tags.

Parameters

- **appid** – application name as defined in the APPID table.
- **name** – extended data list name
- **tags** – iterable of (code, value) tuples or *DXFTag*

discard_xdata_list (*appid*: str, *name*: str) → None
 Discard tag list *name* for extended data *appid*. Does not raise an exception if no extended data for *appid* or no tag list *name* exist.

replace_xdata_list (*appid*: str, *name*: str, *tags*: Iterable)
 Replaces tag list *name* for existing extended data *appid* by *tags*. Appends new list if tag list *name* do not exist, but raises DXFValueError if extended data *appid* do not exist.

Parameters

- **appid** – application name as defined in the APPID table.
- **name** – extended data list name
- **tags** – iterable of (code, value) tuples or *DXFTag*

Raises DXFValueError – no extended data for *appid* found

has_reactors () → bool
 Returns True if entity has reactors.

get_reactors() → List[str]
Returns associated reactors as list of handles.

set_reactors(handles: Iterable[str]) → None
Set reactors as list of handles.

append_reactor_handle(handle: str) → None
Append *handle* to reactors.

discard_reactor_handle(handle: str) → None
Discard *handle* from reactors. Does not raise an exception if *handle* does not exist.

DXF Graphic Entity Base Class

Common base class for all graphical DXF entities.

This entities resides in entity spaces like *Modelspace*, any *Paperspace* or *BlockLayout*.

Subclass of `ezdxf.entities.DXFEntity`

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

class ezdxf.entities.DXFGraphic

rgb
Get/set DXF attribute *dxf.true_color* as (r, g, b) tuple, returns None if attribute *dxf.true_color* is not set.

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

This is the recommend method to get/set RGB values, when ever possible do not use the DXF low level attribute *dxf.true_color*.

transparency

Get/set transparency value as float. Value range 0 to 1, where 0 means entity is opaque and 1 means entity is 100% transparent (invisible). This is the recommend method to get/set transparency values, when ever possible do not use the DXF low level attribute *DXFGraphic.dxf.transparency*

This attribute requires DXF R2004 or later, returns 0 for prior DXF versions and raises *DXFAttributeError* for setting *transparency* in older DXF versions.

ocs() → OCS

Returns object coordinate system (*OCS*) for 2D entities like *Text* or *Circle*, returns a pass-through OCS for entities without OCS support.

get_layout() → BaseLayout

Returns the owner layout or returns None if entity is not assigned to any layout.

unlink_from_layout() → None

Unlink entity from associated layout. Does nothing if entity is already unlinked.

It is more efficient to call the *unlink_entity()* method of the associated layout, especially if you have to unlink more than one entity.

copy_to_layout (*layout: BaseLayout*) → DXFEntity

Copy entity to another *layout*, returns new created entity as *DXFEntity* object. Copying between different DXF drawings is not supported.

Parameters **layout** – any layout (model space, paper space, block)

Raises `DXFStructureError` – for copying between different DXF drawings

move_to_layout (*layout: BaseLayout, source: BaseLayout=None*)

Move entity from model space or a paper space layout to another layout. For block layout as source, the block layout has to be specified. Moving between different DXF drawings is not supported.

Parameters

- **layout** – any layout (model space, paper space, block)
- **source** – provide source layout, faster for DXF R12, if entity is in a block layout

Raises `DXFStructureError` – for moving between different DXF drawings

graphic_properties () → Dict[KT, VT]

Returns the important common properties layer, color, linetype, linewidth, ltscale, true_color and color_name as *dxfattribs* dict.

has_hyperlink () → bool

Returns True if entity has an attached hyperlink.

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

Returns hyperlink, description and location.

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

Set hyperlink of an entity.

transform (*t: Matrix44*) → DXFGraphic

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

Parameters **m** – 4x4 transformation matrix (`ezdxf.math.Matrix44`)

translate (*dx: float, dy: float, dz: float*) → DXFGraphic

Translate entity inplace about *dx* in x-axis, *dy* in y-axis and *dz* in z-axis, returns *self* (floating interface).

Basic implementation uses the `transform()` interface, subclasses may have faster implementations.

scale (*sx: float, sy: float, sz: float*) → DXFGraphic

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

scale_uniform (*s: float*) → DXFGraphic

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

rotate_x (*angle: float*) → DXFGraphic

Rotate entity inplace about x-axis, returns *self* (floating interface).

Parameters **angle** – rotation angle in radians

rotate_y (*angle: float*) → DXFGraphic

Rotate entity inplace about y-axis, returns *self* (floating interface).

Parameters **angle** – rotation angle in radians

rotate_z (*angle: float*) → DXFGraphic

Rotate entity inplace about z-axis, returns *self* (floating interface).

Parameters **angle** – rotation angle in radians

rotate_axis (*axis: Vec3, angle: float*) → DXFGraphic

Rotate entity inplace about vector *axis*, returns *self* (floating interface).

Parameters

- **axis** – rotation axis as tuple or Vec3
- **angle** – rotation angle in radians

Common graphical DXF attributes

DXFGraphic.dxf.layer

Layer name as string; default = '0'

DXFGraphic.dxf.linetype

Linetype as string, special names 'BYLAYER', 'BYBLOCK'; default value is 'BYLAYER'

DXFGraphic.dxf.color

AutoCAD Color Index (ACI), default = 256

Constants defined in `ezdxf.1ldxf.const`

0	BYBLOCK
256	BYLAYER
257	BYOBJECT

DXFGraphic.dxf.lineweight

Line weight in mm times 100 (e.g. 0.13mm = 13). There are fixed valid lineweights which are accepted by AutoCAD, other values prevents AutoCAD from loading the DXF document, BricsCAD isn't that picky. (requires DXF R2000)

Constants defined in `ezdxf.1ldxf.const`

-1	LINEWEIGHT_BYLAYER
-2	LINEWEIGHT_BYBLOCK
-3	LINEWEIGHT_DEFAULT

Valid DXF lineweights stored in `VALID_DXF_LINEWEIGHTS`: 0, 5, 9, 13, 15, 18, 20, 25, 30, 35, 40, 50, 53, 60, 70, 80, 90, 100, 106, 120, 140, 158, 200, 211

DXFGraphic.dxf.ltSCALE

Line type scale as float; default = 1.0 (requires DXF R2000)

DXFGraphic.dxf.invisible

1 for invisible, 0 for visible; default = 0 (requires DXF R2000)

DXFGraphic.dxf.paperSpace

0 for entity resides in modelspace or a block, 1 for paperspace, this attribute is set automatically by adding an entity to a layout (feature for experts); default = 0

DXFGraphic.dxf.extrusion

Extrusion direction as 3D vector; default = (0, 0, 1)

DXFGraphic.dxf.thickness

Entity thickness as float; default = 0.0 (requires DXF R2000)

DXFGraphic.dxf.true_color

True color value as int 0x00RRGGBB, use `DXFGraphic.rgb` to get/set true color values as (r, g, b) tuples. (requires DXF R2004)

DXFGraphic.dxf.color_name

Color name as string. (requires DXF R2004)

DXFGraphic.dxf.transparency

Transparency value as int, $0x020000TT$ $0x00 = 100\%$ transparent / $0xFF = \text{opaque}$, special value $0x01000000$ means transparency by block. An unset transparency value means transparency by layer. Use `DXFGraphic.transparency` to get/set transparency as float value, and the properties `DXFGraphic.is_transparency_by_block` and `DXFGraphic.is_transparency_by_layer` to check special cases.

(requires DXF R2004)

DXFGraphic.dxf.shadow_mode

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

(requires DXF R2007)

Face3d

A 3DFACE ([DXF Reference](#)) is real 3D solid filled triangle or quadrilateral. Access vertices by name (`entity.dxf.vtx0 = (1.7, 2.3)`) or by index (`entity[0] = (1.7, 2.3)`).

Subclass of	<code>ezdxf.entities.DXFGraphic</code>
DXF type	'3DFACE'
Factory function	<code>ezdxf.layouts.BaseLayout.add_3dface()</code>
Inherited DXF attributes	<i>Common graphical DXF attributes</i>

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

class ezdxf.entities.Face3d

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

dx.dxf.vtx0

Location of 1. vertex (3D Point in [WCS](#))

dx.dxf.vtx1

Location of 2. vertex (3D Point in [WCS](#))

dx.dxf.vtx2

Location of 3. vertex (3D Point in [WCS](#))

dx.dxf.vtx3

Location of 4. vertex (3D Point in [WCS](#))

dx.dxf.invisible_edge

invisible edge flag (int, default=0)

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

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

transform(*m*: Matrix44) → Face3d

Transform the 3DFACE entity by transformation matrix *m* inplace.

wcs_vertices(*close*: bool=False) → List[Vec3]

Returns WCS vertices, if argument *close* is True, last vertex == first vertex.

returns 4 vertices when close=False and 5 vertices when close=True. Some edges may have 0 length.

Compatibility interface to SOLID and TRACE. The 3DFACE entity returns already WCS vertices.

Solid3d

3DSOLID ([DXF Reference](#)) created by an ACIS based geometry kernel provided by the Spatial Corp.

See also:

*Ezdx*f will never create or interpret ACIS data, for more information see the FAQ: [How to add/edit ACIS based entities like 3DSOLID, REGION or SURFACE?](#)

Subclass of	<code>ezdxf.entities.Body</code>
DXF type	'3DSOLID'
Factory function	<code>ezdxf.layouts.BaseLayout.add_3dsolid()</code>
Inherited DXF attributes	Common graphical DXF attributes
Required DXF version	DXF R2000 ('AC1015')

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

class ezdxf.entities.**Solid3d**

Same attributes and methods as parent class `Body`.

`dx.dxf.history_handle`

Handle to history object.

Arc

ARC ([DXF Reference](#)) center at location `dx.dxf.center` and radius of `dx.dxf.radius` from `dx.dxf.start_angle` to `dx.dxf.end_angle`. ARC goes always from `dx.dxf.start_angle` to `dx.dxf.end_angle` in counter clockwise orientation around the `dx.dxf.extrusion` vector, which is (0, 0, 1) by default and the usual case for 2D arcs.

Subclass of	<code>ezdxf.entities.Circle</code>
DXF type	'ARC'
Factory function	<code>ezdxf.layouts.BaseLayout.add_arc()</code>
Inherited DXF attributes	Common graphical DXF attributes

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

```
class ezdxf.entities.Arc
```

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

dxfs.radius
Radius of arc (float)

dxfs.start_angle
Start angle in degrees (float)

dxfs.end_angle
End angle in degrees (float)

start_point
Returns the start point of the arc in WCS, takes OCS into account.

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

angles (num: int) → Iterable[float]
Returns num angles from start- to end angle in degrees in counter clockwise order.
All angles are normalized in the range from [0, 360].

flattening (sagitta: float) → Iterable[Vertex]
Approximate the arc by vertices in WCS, argument segment is the max. distance from the center of an arc segment to the center of its chord. Yields *Vec2* objects for 2D arcs and *Vec3* objects for 3D arcs.
New in version 0.15.

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

to_ellipse (replace=True) → Ellipse
Convert CIRCLE/ARC to an *Ellipse* entity.
Adds the new ELLIPSE entity to the entity database and to the same layout as the source entity.
Parameters **replace** – replace (delete) source entity by ELLIPSE entity if True

to_spline (replace=True) → Spline
Convert CIRCLE/ARC to a *Spline* entity.
Adds the new SPLINE entity to the entity database and to the same layout as the source entity.
Parameters **replace** – replace (delete) source entity by SPLINE entity if True

construction_tool () → ConstructionArc
Returns 2D construction tool *ezdxf.math.ConstructionArc*, ignoring the extrusion vector.

apply_construction_tool (arc: ConstructionArc) → Arc
Set ARC data from construction tool *ezdxf.math.ConstructionArc*, will not change the extrusion vector.

Body

BODY ([DXF Reference](#)) created by an ACIS based geometry kernel provided by the [Spatial Corp.](#)

See also:

*Ezdx*f will never create or interpret ACIS data, for more information see the FAQ: [How to add/edit ACIS based entities like 3DSOLID, REGION or SURFACE?](#)

Subclass of	<code>ezdxf.entities.DXFGraphic</code>
DXF type	'BODY'
Factory function	<code>ezdxf.layouts.BaseLayout.add_body()</code>
Inherited DXF attributes	Common graphical DXF attributes
Required DXF version	DXF R2000 ('AC1015')

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

`class ezdxf.entities.Body`

`dx.f.version`

Modeler format version number, default value is 1

`dx.f.flags`

Require DXF R2013.

`dx.f.uid`

Require DXF R2013.

`acis_data`

Get/Set ACIS text data as list of strings for DXF R2000 to R2010 and binary encoded ACIS data for DXF R2013 and later as list of bytes.

`has_binary_data`

Returns True if ACIS data is of type `List[bytes]`, False if data is of type `List[str]`.

`tostring() → str`

Returns ACIS data as one string for DXF R2000 to R2010.

`tobytes() → bytes`

Returns ACIS data as joined bytes for DXF R2013 and later.

`set_text(text: str, sep: str = '\n') → None`

Set ACIS data from one string.

Circle

CIRCLE ([DXF Reference](#)) center at location `dx.f.center` and radius of `dx.f.radius`.

Subclass of	<code>ezdxf.entities.DXFGraphic</code>
DXF type	'CIRCLE'
Factory function	<code>ezdxf.layouts.BaseLayout.add_circle()</code>
Inherited DXF attributes	Common graphical DXF attributes

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

```
class ezdxf.entities.Circle

    dxf.center
        Center point of circle (2D/3D Point in OCS)
    dxf.radius
        Radius of circle (float)
    vertices (angle:Iterable[float]) → Iterable[Vec3]
        Yields vertices of the circle for iterable angles in WCS.
        Parameters angles – iterable of angles in OCS as degrees, angle goes counter clockwise
        around the extrusion vector, OCS x-axis = 0 deg.
    flattening (sagitta: float) → Iterable[Vec3]
        Approximate the circle by vertices in WCS, argument sagitta is the max. distance from the center of an
        arc segment to the center of its chord. Returns a closed polygon: start vertex == end vertex!
        Yields always Vec3 objects.
        New in version 0.15.
    transform (m: Matrix44) → Circle
        Transform the CIRCLE entity by transformation matrix m inplace.
        Raises NonUniformScalingError () for non uniform scaling.
    translate (dx: float, dy: float, dz: float) → Circle
        Optimized CIRCLE/ARC translation about dx in x-axis, dy in y-axis and dz in z-axis, returns self (floating
        interface).
    to_ellipse (replace=True) → Ellipse
        Convert CIRCLE/ARC to an Ellipse entity.
        Adds the new ELLIPSE entity to the entity database and to the same layout as the source entity.
        Parameters replace – replace (delete) source entity by ELLIPSE entity if True
    to_spline (replace=True) → Spline
        Convert CIRCLE/ARC to a Spline entity.
        Adds the new SPLINE entity to the entity database and to the same layout as the source entity.
        Parameters replace – replace (delete) source entity by SPLINE entity if True
```

Dimension

The DIMENSION entity ([DXF Reference](#)) represents several types of dimensions in many orientations and alignments. The basic types of dimensioning are linear, radial, angular, ordinate, and arc length.

For more information about dimensions see the online help from AutoDesk: [About the Types of Dimensions](#)

Important: The DIMENSION entity is reused to create dimensional constraints, such entities do not have an associated geometrical block nor a dimension type group code (2) and reside on layer *ADSK_CONSTRAINTS. Use property `Dimension.is_dimensional_constraint` to check for this objects. Dimensional constraints are not documented in the DXF reference and not supported by *ezdxf*.

See also:

- [Tutorial for Linear Dimensions](#)
- [Tutorial for Radius Dimensions](#)
- [Tutorial for Diameter Dimensions](#)
- [Tutorial for Angular Dimensions](#)
- [Tutorial for Ordinate Dimensions](#)

Subclass of	ezdxf.entities.DXFGraphic
DXF type	'DIMENSION'
factory function	see table below
Inherited DXF attributes	Common graphical DXF attributes

Factory Functions

Linear and Rotated Dimension (DXF)	add_linear_dim()
Aligned Dimension (DXF)	add_aligned_dim()
Angular Dimension (DXF)	add_angular_dim_21()
Angular 3P Dimension (DXF)	add_angular_dim_3p()
Angular Dimension by center, radius, angles	add_angular_dim_cra()
Angular Dimension by ConstructionArc	add_angular_dim_arc()
Diameter Dimension (DXF)	add_diameter_dim()
Radius Dimension (DXF)	add_radius_dim()
Ordinate Dimension (DXF)	add_ordinate_dim() (not implemented)

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

class `ezdxf.entities.Dimension`

There is only one `Dimension` class to represent all different dimension types.

dxfs.version

Version number: 0 = R2010. (int, DXF R2010)

dxfs.geometry

Name of the BLOCK that contains the entities that make up the dimension picture.

For AutoCAD this graphical representation is mandatory, else AutoCAD will not open the DXF drawing. BricsCAD will render the DIMENSION entity by itself, if the graphical representation is not present, but uses the BLOCK instead of rendering, if it is present.

dxfs.dimstyle

Dimension style (`DimStyle`) name as string.

dxfs.dimtype

Values 0-6 are integer values that represent the dimension type. Values 32, 64, and 128 are bit values, which are added to the integer values.

0	Linear and Rotated Dimension (DXF)
1	Aligned Dimension (DXF)
2	Angular Dimension (DXF)
3	Diameter Dimension (DXF)
4	Radius Dimension (DXF)
5	Angular 3P Dimension (DXF)
6	Ordinate Dimension (DXF)
8	subclass <code>ezdxf.entities.ArcDimension</code> introduced in DXF R2004
32	Indicates that graphical representation geometry is referenced by this dimension only. (always set in DXF R13 and later)
64	Ordinate type. This is a bit value (bit 7) used only with integer value 6. If set, ordinate is <i>X-type</i> ; if not set, ordinate is <i>Y-type</i>
128	This is a bit value (bit 8) added to the other dimtype values if the dimension text has been positioned at a user-defined location rather than at the default location

dx.dxf.defpoint

Definition point for all dimension types. (3D Point in [WCS](#))

Linear and rotated dimension: dx.dxf.defpoint specifies the dimension line location.

Arc and angular dimension: dx.dxf.defpoint and dx.dxf.defpoint4 specify the endpoints of the line used to determine the second extension line.

dx.dxf.defpoint2

Definition point for linear and angular dimensions. (3D Point in [WCS](#))

Linear and rotated dimension: The dx.dxf.defpoint2 specifies the start point of the first extension line.

Arc and angular dimension: The dx.dxf.defpoint2 and dx.dxf.defpoint3 specify the endpoints of the line used to determine the first extension line.

dx.dxf.defpoint3

Definition point for linear and angular dimensions. (3D Point in [WCS](#))

Linear and rotated dimension: The dx.dxf.defpoint3 specifies the start point of the second extension line.

Arc and angular dimension: The dx.dxf.defpoint2 and dx.dxf.defpoint3 specify the endpoints of the line used to determine the first extension line.

dx.dxf.defpoint4

Definition point for diameter, radius, and angular dimensions. (3D Point in [WCS](#))

Arc and angular dimension: dx.dxf.defpoint and dx.dxf.defpoint4 specify the endpoints of the line used to determine the second extension line.

dx.dxf.defpoint5

Point defining dimension arc for angular dimensions, specifies the location of the dimension line arc. (3D Point in [OCS](#))

dx.dxf.angle

Angle of linear and rotated dimensions in degrees. (float)

dx.dxf.leader_length

Leader length for radius and diameter dimensions. (float)

dx.dxf.text_midpoint

Middle point of dimension text. (3D Point in [OCS](#))

dx_f.insert

Insertion point for clones of a linear dimensions. (3D Point in *OCS*)

This value translates the content of the associated anonymous block for cloned linear dimensions, similar to the `insert` attribute of the [Insert](#) entity.

dx_f.attachment_point

Text attachment point (int, DXF R2000), default value is 5.

1	Top left
2	Top center
3	Top right
4	Middle left
5	Middle center
6	Middle right
7	Bottom left
8	Bottom center
9	Bottom right

dx_f.line_spacing_style

Dimension text line-spacing style (int, DXF R2000), default value is 1.

1	At least (taller characters will override)
2	Exact (taller characters will not override)

dx_f.line_spacing_factor

Dimension text-line spacing factor. (float, DXF R2000)

Percentage of default (3-on-5) line spacing to be applied. Valid values range from 0.25 to 4.00.

dx_f.actual_measurement

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

dx_f.text

Dimension text explicitly entered by the user (str), default value is an empty string.

If empty string or '`<>`', the dimension measurement is drawn as the text, if ' ' (one blank space), the text is suppressed. Anything else is drawn as the text.

dx_f.oblique_angle

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

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

dx_f.text_rotation

Defines is the rotation angle of the dimension text away from its default orientation (the direction of the dimension line). (float)

dx_f.horizontal_direction

Indicates the horizontal direction for the dimension entity (float).

This attribute determines the orientation of dimension text and lines for horizontal, vertical, and rotated linear dimensions. This value is the negative of the angle in the OCS xy-plane between the dimension line and the OCS x-axis.

get_dim_style() → DimStyle

Returns the associated [DimStyle](#) entity.

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

get_measurement() → Union[float, ezdxf.math._vector.Vec3]
 Returns the actual dimension measurement in *WCS* units, no scaling applied for linear dimensions. Returns angle in degrees for angular dimension from 2 lines and angular dimension from 3 points. Returns vector from origin to feature location for ordinate dimensions.

override() → DimStyleOverride
 Returns the *DimStyleOverride* object.

render()
 Render graphical representation as anonymous block.

transform(m: Matrix44) → Dimension
 Transform the DIMENSION entity by transformation matrix *m* inplace.
 Raises NonUniformScalingError() for non uniform scaling.

virtual_entities() → Iterable[DXFGraphic]
 Yields ‘virtual’ parts of DIMENSION as basic DXF entities like LINE, ARC or TEXT.
 This entities are located at the original positions, but are not stored in the entity database, have no handle and are not assigned to any layout.

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

Parameters **target_layout** – target layout for DXF parts, None for same layout as source entity.

DimStyleOverride

All of the *DimStyle* attributes can be overridden for each *Dimension* entity individually.

The *DimStyleOverride* class manages all the complex dependencies between *DimStyle* and *Dimension*, the different features of all DXF versions and the rendering process to create the *Dimension* picture as BLOCK, which is required for AutoCAD.

```
class ezdxf.entities.DimStyleOverride

    dimension
        Base Dimension entity.

    dimstyle
        By dimension referenced DimStyle entity.

    dimstyle_attribs
        Contains all overridden attributes of dimension, as a dict with DimStyle attribute names as keys.

    __getitem__(key: str) → Any
        Returns DIMSTYLE attribute key, see also get().

    __setitem__(key: str, value: Any) → None
        Set DIMSTYLE attribute key in dimstyle_attribs.
```

__delitem__ (*key: str*) → None
Deletes DIMSTYLE attribute *key* from *dimstyle_attrs*, ignores KeyErrors silently.

get (*attribute: str, default: Any = None*) → Any
Returns DIMSTYLE *attribute* from override dict *dimstyle_attrs* or base *DimStyle*.
Returns *default* value for attributes not supported by DXF R12. This is a hack to use the same algorithm to render DXF R2000 and DXF R12 DIMENSION entities. But the DXF R2000 attributes are not stored in the DXF R12 file! Does not catch invalid attributes names! Look into debug log for ignored DIMSTYLE attributes.

pop (*attribute: str, default: Any = None*) → Any
Returns DIMSTYLE *attribute* from override dict *dimstyle_attrs* and removes this *attribute* from override dict.

update (*attribs: dict*) → None
Update override dict *dimstyle_attrs*.

Parameters attribs – dict of DIMSTYLE attributes

commit () → None
Writes overridden DIMSTYLE attributes into ACAD:DSTYLE section of XDATA of the DIMENSION entity.

get_arrow_names () → Tuple[str, str]
Get arrow names as strings like 'ARCHTICK'.
Returns tuple of [dimblk1, dimblk2]
Return type Tuple[str, str]

set_arrows (*blk: str = None, blk1: str = None, blk2: str = None, ldrblk: str = None, size: float = None*) → None
Set arrows or user defined blocks and disable oblique stroke as tick.
Parameters

- **blk** – defines both arrows at once as name str or user defined block
- **blk1** – defines left arrow as name str or as user defined block
- **blk2** – defines right arrow as name str or as user defined block
- **ldrblk** – defines leader arrow as name str or as user defined block
- **size** – arrow size in drawing units

set_tick (*size: float = 1*) → None
Use oblique stroke as tick, disables arrows.
Parameters size – arrow size in daring units

set_text_align (*halign: str = None, valign: str = None, vshift: float = None*) → None
Set measurement text alignment, *halign* defines the horizontal alignment, *valign* defines the vertical alignment, *above1* and *above2* means above extension line 1 or 2 and aligned with extension line.
Parameters

- **halign** – *left, right, center, above1, above2*, requires DXF R2000+
- **valign** – *above, center, below*
- **vshift** – vertical text shift, if *valign* is *center*; >0 shift upward, <0 shift downwards

```
set_tolerance(upper: float, lower: float = None, hfactor: float = None, align: ezdxf.enums.MTextLineAlignment = None, dec: int = None, leading_zeros: bool = None, trailing_zeros: bool = None) → None
```

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

Parameters

- **upper** – upper tolerance value
- **lower** – lower tolerance value, if None same as upper
- **hfactor** – tolerance text height factor in relation to the dimension text height
- **align** – tolerance text alignment enum `ezdxf.enums.MTextLineAlignment`
- **dec** – Sets the number of decimal places displayed
- **leading_zeros** – suppress leading zeros for decimal dimensions if False
- **trailing_zeros** – suppress trailing zeros for decimal dimensions if False

Changed in version 0.17.2: argument `align` as enum `ezdxf.enums.MTextLineAlignment`

```
set_limits(upper: float, lower: float, hfactor: float = None, dec: int = None, leading_zeros: bool = None, trailing_zeros: bool = None) → None
```

Set limits text format, upper and lower limit values, text height factor, number of decimal places or leading and trailing zero suppression.

Parameters

- **upper** – upper limit value added to measurement value
- **lower** – lower lower value subtracted from measurement value
- **hfactor** – limit text height factor in relation to the dimension text height
- **dec** – Sets the number of decimal places displayed, required DXF R2000+
- **leading_zeros** – suppress leading zeros for decimal dimensions if False, required DXF R2000+
- **trailing_zeros** – suppress trailing zeros for decimal dimensions if False, required DXF R2000+

```
set_text_format(prefix: str = "", postfix: str = "", rnd: float = None, dec: int = None, sep: str = None, leading_zeros: bool = None, trailing_zeros: bool = None) → None
```

Set dimension text format, like prefix and postfix string, rounding rule and number of decimal places.

Parameters

- **prefix** – dimension text prefix text as string
- **postfix** – dimension text postfix text as string
- **rnd** – Rounds all dimensioning distances to the specified value, for instance, if DIMRND is set to 0.25, all distances round to the nearest 0.25 unit. If you set DIMRND to 1.0, all distances round to the nearest integer.
- **dec** – Sets the number of decimal places displayed for the primary units of a dimension. requires DXF R2000+
- **sep** – “.” or “,” as decimal separator
- **leading_zeros** – suppress leading zeros for decimal dimensions if False
- **trailing_zeros** – suppress trailing zeros for decimal dimensions if False

set_dimline_format (*color: int = None, linetype: str = None, linewidth: int = None, extension: float = None, disable1: bool = None, disable2: bool = None*)
Set dimension line properties

Parameters

- **color** – color index
- **linetype** – linetype as string
- **linewidth** – line weight as int, 13 = 0.13mm, 200 = 2.00mm
- **extension** – extension length
- **disable1** – True to suppress first part of dimension line
- **disable2** – True to suppress second part of dimension line

set_extline_format (*color: int = None, linewidth: int = None, extension: float = None, offset: float = None, fixed_length: float = None*)
Set common extension line attributes.

Parameters

- **color** – color index
- **linewidth** – line weight as int, 13 = 0.13mm, 200 = 2.00mm
- **extension** – extension length above dimension line
- **offset** – offset from measurement point
- **fixed_length** – set fixed length extension line, length below the dimension line

set_extline1 (*linetype: str = None, disable=False*)
Set extension line 1 attributes.

Parameters

- **linetype** – linetype for extension line 1
- **disable** – disable extension line 1 if True

set_extline2 (*linetype: str = None, disable=False*)
Set extension line 2 attributes.

Parameters

- **linetype** – linetype for extension line 2
- **disable** – disable extension line 2 if True

set_text (*text: str = '<>'*) → None
Set dimension text.

- *text* = “” to suppress dimension text
- *text* = “” or “<>” to use measured distance as dimension text
- else use “text” literally

shift_text (*dh: float, dv: float*) → None
Set relative text movement, implemented as user location override without leader.

Parameters

- **dh** – shift text in text direction
- **dv** – shift text perpendicular to text direction

set_location (*location: Vertex, leader=False, relative=False*) → None

Set text location by user, special version for linear dimensions, behaves for other dimension types like `user_location_override()`.

Parameters

- **location** – user defined text location (Vertex)
- **leader** – create leader from text to dimension line
- **relative** – *location* is relative to default location.

user_location_override (*location: Vertex*) → None

Set text location by user, *location* is relative to the origin of the UCS defined in the `render()` method or WCS if the *ucs* argument is None.

render (*ucs: UCS = None, discard=False*) → BaseDimensionRenderer

Initiate dimension line rendering process and also writes overridden dimension style attributes into the DSTYLE XDATA section.

For a friendly CAD applications like BricsCAD you can discard the dimension line rendering, because it is done automatically by BricsCAD, if no dimension rendering BLOCK is available and it is likely to get better results as by *ezdxf*.

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

Parameters

- **ucs** – user coordinate system
- **discard** – discard rendering done by *ezdxf* (works with BricsCAD, but not tolerated by AutoCAD)

Returns Rendering object used to render the DIMENSION entity for analytics

Return type BaseDimensionRenderer

ArcDimension

The ARC_DIMENSION entity was introduced in DXF R2004 and is **not** documented in the DXF reference.

See also:

[Tutorial for Arc Dimensions](#)

Subclass of	<code>ezdxf.entities.Dimension</code>
DXF type	'ARC_DIMENSION'
factory function	<ul style="list-style-type: none"> • <code>add_arc_dim_3p()</code> • <code>add_arc_dim_cra()</code> • <code>add_arc_dim_arc()</code>
Inherited DXF attributes	Common graphical DXF attributes
Required DXF version	R2004 / AC1018

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

```
class ezdxf.entities.ArcDimension

    dxf.defpoint2
        start point of first extension line in OCS

    dxf.defpoint3
        start point of second extension line in OCS

    dxf.defpoint4
        center point of arc in OCS

    dxf.start_angle
    dxf.end_angle
    dxf.is_partial
    dxf.has_leader
    dxf.leader_point1
    dxf.leader_point2

    dimtype
        Returns always 8.
```

Ellipse

ELLIPSE ([DXF Reference](#)) with center point at location `dxf.center` and a major axis `dxf.major_axis` as vector. `dxf.ratio` is the ratio of minor axis to major axis. `dxf.start_param` and `dxf.end_param` defines the starting- and the end point of the ellipse, a full ellipse goes from 0 to 2π . The ellipse goes from starting- to end param in counter clockwise direction.

`dxf.extrusion` is supported, but does not establish an *OCS*, but creates an 3D entity by extruding the base ellipse in direction of the `dxf.extrusion` vector.

Subclass of	ezdxf.entities.DXFGraphic
DXF type	'ELLIPSE'
factory function	add_ellipse()
Inherited DXF attributes	Common graphical DXF attributes
Required DXF version	DXF R2000 ('AC1015')

class ezdxf.entities.Ellipse

```
    dxf.center
        Center point of circle (2D/3D Point in WCS)

    dxf.major_axis
        Endpoint of major axis, relative to the dxf.center (Vec3), default value is (1, 0, 0).

    dxf.ratio
        Ratio of minor axis to major axis (float), has to be in range from 0.000001 to 1, default value is 1.

    dxf.start_param
        Start parameter (float), default value is 0.

    dxf.end_param
        End parameter (float), default value is  $2\pi$ .
```

start_point

Returns the start point of the ellipse in WCS.

end_point

Returns the end point of the ellipse in WCS.

minor_axis

Returns the minor axis of the ellipse as Vec3 in WCS.

construction_tool() → ConstructionEllipse

Returns construction tool `ezdxf.math.ConstructionEllipse`.

apply_construction_tool(*e*: ConstructionEllipse) → Ellipse

Set ELLIPSE data from construction tool `ezdxf.math.ConstructionEllipse`.

vertices(*params*: Iterable[float]) → Iterable[Vec3]

Yields vertices on ellipse for iterable *params* in WCS.

Parameters **params** – param values in the range from 0 to 2π in radians, param goes counter clockwise around the extrusion vector, major_axis = local x-axis = 0 rad.

flattening(*distance*: float, *segments*: int = 8) → Iterable[Vec3]

Adaptive recursive flattening. The argument *segments* is the minimum count of approximation segments, if the distance from the center of the approximation segment to the curve is bigger than *distance* the segment will be subdivided. Returns a closed polygon for a full ellipse: start vertex == end vertex.

Parameters

- **distance** – maximum distance from the projected curve point onto the segment chord.
- **segments** – minimum segment count

New in version 0.15.

params(*num*: int) → Iterable[float]

Returns *num* params from start- to end param in counter clockwise order.

All params are normalized in the range from [0, 2π).

transform(*m*: Matrix44) → Ellipse

Transform the ELLIPSE entity by transformation matrix *m* inplace.

translate(*dx*: float, *dy*: float, *dz*: float) → Ellipse

Optimized ELLIPSE translation about *dx* in x-axis, *dy* in y-axis and *dz* in z-axis, returns *self* (floating interface).

to_spline(*replace=True*) → Spline

Convert ELLIPSE to a `Spline` entity.

Adds the new SPLINE entity to the entity database and to the same layout as the source entity.

Parameters **replace** – replace (delete) source entity by SPLINE entity if True

classmethod from_arc(*entity*: DXFGraphic) → Ellipse

Create a new ELLIPSE entity from ARC or CIRCLE entity.

The new SPLINE entity has no owner, no handle, is not stored in the entity database nor assigned to any layout!

Hatch

The HATCH entity ([DXF Reference](#)) fills an enclosed area defined by one or more boundary paths with a hatch pattern, solid fill, or gradient fill.

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

There are two different hatch pattern default scaling, depending on the HEADER variable \$MEASUREMENT, one for ISO measurement (m, cm, mm, ...) and one for imperial measurement (in, ft, yd, ...).

Starting with *ezdxf* v0.15 the default scaling for predefined hatch pattern will be chosen according this measurement setting in the HEADER section, this replicates the behavior of BricsCAD and other CAD applications. *ezdxf* uses the ISO pattern definitions as a base line and scales this pattern down by factor 1/25.6 for imperial measurement usage. The pattern scaling is independent from the drawing units of the document defined by the HEADER variable \$INSUNITS.

Prior to *ezdxf* v0.15 the default scaling was always the ISO measurement scaling, no matter which value \$MEASUREMENT had.

See also:

[Tutorial for Hatch and DXF Units](#)

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

Boundary paths helper classes

Path manager: `BoundaryPaths`

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

Pattern and gradient helper classes

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

`class ezdxf.entities.Hatch`

```
    dxf.pattern_name
        Pattern name as string
    dxf.solid_fill
```

1	solid fill, better use: <code>Hatch.set_solid_fill()</code>
0	pattern fill, better use: <code>Hatch.set_pattern_fill()</code>

dx_f.associative

1	associative hatch
0	not associative hatch

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

dx_f.hatch_style

0	normal
1	outer
2	ignore

(search AutoCAD help for more information)

dx_f.pattern_type

0	user
1	predefined
2	custom

dx_f.pattern_angle

Actual pattern angle in degrees (float). Changing this value does not rotate the pattern, use [set_pattern_angle\(\)](#) for this task.

dx_f.pattern_scale

Actual pattern scaling factor (float). Changing this value does not scale the pattern use [set_pattern_scale\(\)](#) for this task.

dx_f.pattern_double

1 = double pattern size else 0. (int)

dx_f.n_seed_points

Count of seed points (better user: [get_seed_points\(\)](#))

dx_f.elevation

Z value represents the elevation height of the *OCS*. (float)

paths

BoundaryPaths object.

pattern

Pattern object.

gradient

Gradient object.

seeds

List of (x, y) tuples.

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

Setup pattern definition by a list of definition lines and a definition line is a 4-tuple (angle, base_point, offset, dash_length_items), the pattern definition should be designed for scaling factor 1 and angle 0.

- angle: line angle in degrees

- base-point: 2-tuple (x, y)
- offset: 2-tuple (dx, dy)
- dash_length_items: list of dash items (item > 0 is a line, item < 0 is a gap and item == 0.0 is a point)

Parameters

- **lines** – list of definition lines
- **factor** – pattern scaling factor
- **angle** – rotation angle in degrees

set_pattern_scale (*scale*: float) → None

Set scaling of pattern definition to *scale*.

Starts always from the original base scaling, `set_pattern_scale(1)` reset the pattern scaling to the original appearance as defined by the pattern designer, but only if the the pattern attribute `dxf.pattern_scale` represents the actual scaling, it is not possible to recreate the original pattern scaling from the pattern definition itself.

Parameters **scale** – pattern scaling factor

set_pattern_angle (*angle*: float) → None

Set rotation of pattern definition to *angle* in degrees.

Starts always from the original base rotation 0, `set_pattern_angle(0)` reset the pattern rotation to the original appearance as defined by the pattern designer, but only if the the pattern attribute `dxf.pattern_angle` represents the actual rotation, it is not possible to recreate the original rotation from the pattern definition itself.

Parameters **angle** – rotation angle in degrees

set_solid_fill (*color*: int = 7, *style*: int = 1, *rgb*: RGB = None)

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

Parameters

- **color** – [AutoCAD Color Index \(ACI\)](#), (0 = BYBLOCK; 256 = BYLAYER)
- **style** – hatch style (0 = normal; 1 = outer; 2 = ignore)
- **rgb** – true color value as (r, g, b)-tuple - has higher priority than *color*. True color support requires DXF R2000.

set_pattern_fill (*name*: str, *color*: int = 7, *angle*: float = 0.0, *scale*: float = 1.0, *double*: int = 0, *style*: int = 1, *pattern_type*: int = 1, *definition*=None) → None

Set `Hatch` and `MPolygon` to pattern fill mode. Removes all gradient related data. The pattern definition should be designed for scaling factor 1. Predefined hatch pattern like “ANSI33” are scaled according to the HEADER variable \$MEASUREMENT for ISO measurement (m, cm, ...), or imperial units (in, ft, ...), this replicates the behavior of BricsCAD.

Parameters

- **name** – pattern name as string
- **color** – pattern color as [AutoCAD Color Index \(ACI\)](#)
- **angle** – angle of pattern fill in degrees
- **scale** – pattern scaling as float
- **double** – double size flag
- **style** – hatch style (0 = normal; 1 = outer; 2 = ignore)

- **pattern_type** – pattern type (0 = user-defined; 1 = predefined; 2 = custom)
- **definition** – list of definition lines and a definition line is a 4-tuple [angle, base_point, offset, dash_length_items], see `set_pattern_definition()`

set_gradient (`color1: Tuple[int, int, int] = (0, 0, 0), color2: Tuple[int, int, int] = (255, 255, 255), rotation: float = 0.0, centered: float = 0.0, one_color: int = 0, tint: float = 0.0, name: str = 'LINEAR'`) → None

Set `Hatch` and `MPolygon` to gradient fill mode and removes all pattern fill related data. Gradient support requires DXF R2004+. A gradient filled hatch is also a solid filled hatch.

Valid gradient type names are:

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

Parameters

- **color1** – (r, g, b)-tuple for first color, rgb values as int in the range [0, 255]
- **color2** – (r, g, b)-tuple for second color, rgb values as int in the range [0, 255]
- **rotation** – rotation angle in degrees
- **centered** – determines whether the gradient is centered or not
- **one_color** – 1 for gradient from `color1` to tinted `color1`
- **tint** – determines the tinted target `color1` for a one color gradient. (valid range 0.0 to 1.0)
- **name** – name of gradient type, default “LINEAR”

set_seed_points (`points: Iterable[Tuple[float, float]]`) → None

Set seed points, `points` is an iterable of (x, y)-tuples. I don't know why there can be more than one seed point. All points in `OCS` (`Hatch.dxf.elevation` is the Z value)

transform (`m: Matrix44`) → Hatch

Transform entity by transformation matrix `m` inplace.

associate (`path: ezdxf.entities.boundary_paths.AbstractBoundaryPath, entities: Iterable[DXFEntity]`)

Set association from hatch boundary `path` to DXF geometry `entities`.

A HATCH entity can be associative to a base geometry, this association is **not** maintained nor verified by `ezdxf`, so if you modify the base geometry the geometry of the boundary path is not updated and no verification is done to check if the associated geometry matches the boundary path, this opens many possibilities to create invalid DXF files: USE WITH CARE!

remove_association()

Remove associated path elements.

Boundary Paths

The hatch entity is build by different functional path types, this are filter flags for the `Hatch.dxf.hatch_style`:

- EXTERNAL: defines the outer boundary of the hatch
- OUTERMOST: defines the first tier of inner hatch boundaries
- DEFAULT: default boundary path

As you will learn in the next sections, these are more the recommended usage type for the flags, but the fill algorithm doesn't care much about that, for instance an OUTERMOST path doesn't have to be inside the EXTERNAL path.

Island Detection

In general the island detection algorithm works always from outside to inside and alternates filled and unfilled areas. The area between then 1st and the 2nd boundary is filled, the area between the 2nd and the 3rd boundary is unfilled and so on. The different hatch styles defined by the `Hatch.dxf.hatch_style` attribute are created by filtering some boundary path types.

Hatch Style

- HATCH_STYLE_IGNORE: Ignores all paths except the paths marked as EXTERNAL, if there are more than one path marked as EXTERNAL, they are filled in NESTED style. Creates no hatch if no path is marked as EXTERNAL.
- HATCH_STYLE_OUTERMOST: Ignores all paths marked as DEFAULT, remaining EXTERNAL and OUTERMOST paths are filled in NESTED style. Creates no hatch if no path is marked as EXTERNAL or OUTERMOST.
- HATCH_STYLE_NESTED: Use all existing paths.

Hatch Boundary Helper Classes

`class ezdxf.entities.BoundaryPaths`

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

`paths`

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

`external_paths () → Iterable[Union[PolylinePath, EdgePath]]`

Iterable of external paths, could be empty.

`outermost_paths () → Iterable[Union[PolylinePath, EdgePath]]`

Iterable of outermost paths, could be empty.

`default_paths () → Iterable[Union[PolylinePath, EdgePath]]`

Iterable of default paths, could be empty.

`rendering_paths (hatch_style: int) → Iterable[Union[PolylinePath, EdgePath]]`

Iterable of paths to process for rendering, filters unused boundary paths according to the given hatch style:

- NESTED: use all boundary paths
- OUTERMOST: use EXTERNAL and OUTERMOST boundary paths
- IGNORE: ignore all paths except EXTERNAL boundary paths

Yields paths in order of EXTERNAL, OUTERMOST and DEFAULT.

add polyline path (*path_vertices*, *is_closed*=1, *flags*=1) → PolylinePath
Create and add a new [PolylinePath](#) object.

Parameters

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

add edge path (*flags*=1) → EdgePath
Create and add a new [EdgePath](#) object.

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

polyline_to_edge_paths (*just_with_bulge*=True) → None
Convert polyline paths including bulge values to line- and arc edges.

Parameters **just_with_bulge** – convert only polyline paths including bulge values if True

edge_to_polyline_paths (*distance*: float, *segments*: int = 16)
Convert all edge paths to simple polyline paths without bulges.

Parameters

- **distance** – maximum distance from the center of the curve to the center of the line segment between two approximation points to determine if a segment should be subdivided.
- **segments** – minimum segment count per curve

arc_edges_to_ellipse_edges () → None
Convert all arc edges to ellipse edges.

ellipse_edges_to_spline_edges (*num*: int = 32) → None
Convert all ellipse edges to spline edges (approximation).

Parameters **num** – count of control points for a **full** ellipse, partial ellipses have proportional fewer control points but at least 3.

spline_edges_to_line_edges (*factor*: int = 8) → None
Convert all spline edges to line edges (approximation).

Parameters **factor** – count of approximation segments = count of control points x factor

all_to_spline_edges (*num*: int = 64) → None
Convert all bulge, arc and ellipse edges to spline edges (approximation).

Parameters **num** – count of control points for a **full** circle/ellipse, partial circles/ellipses have proportional fewer control points but at least 3.

all_to_line_edges (*num*: int = 64, *spline_factor*: int = 8) → None
Convert all bulge, arc and ellipse edges to spline edges and approximate this splines by line edges.

Parameters

- **num** – count of control points for a **full** circle/ellipse, partial circles/ellipses have proportional fewer control points but at least 3.
- **spline_factor** – count of spline approximation segments = count of control points x *spline_factor*

clear () → None
Remove all boundary paths.

```
class ezdxf.entities.BoundaryPathType
```

POLYLINE

polyline path type

EDGE

edge path type

```
class ezdxf.entities.PolylinePath
```

A polyline as hatch boundary path.

type

Path type as *BoundaryPathType.POLYLINE* enum

path_type_flags

(bit coded flags)

0	default
1	external
2	polyline, will be set by <i>ezdxf</i>
16	outermost

My interpretation of the *path_type_flags*, see also *Tutorial for Hatch*:

- external - path is part of the hatch outer border
- outermost - path is completely inside of one or more external paths
- default - path is completely inside of one or more outermost paths

If there are troubles with AutoCAD, maybe the hatch entity has the *Hatch.dxf.pixel_size* attribute set - delete it del *hatch.dxf.pixel_size* and maybe the problem is solved. *ezdxf* does not use the *Hatch.dxf.pixel_size* attribute, but it can occur in DXF files created by other applications.

is_closed

True if polyline path is closed.

vertices

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

source_boundary_objects

List of handles of the associated DXF entities for associative hatches. There is no support for associative hatches by *ezdxf*, you have to do it all by yourself. (read/write)

set_vertices (*vertices*: *Iterable[Sequence[float]]*, *is_closed*: *bool* = *True*) → None

Set new *vertices* as new polyline path, a vertex has to be a (x, y) or a (x, y, bulge)-tuple.

clear() → None

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

```
class ezdxf.entities.EdgePath
```

Boundary path build by edges. There are four different edge types: *LineEdge*, *ArcEdge*, *EllipseEdge* of *SplineEdge*. Make sure there are no gaps between edges and the edge path must be closed to be recognized as path. AutoCAD is very picky in this regard. *Ezdx* performs no checks on gaps between the edges and does not prevent creating open loops.

Note: *ArcEdge* and *EllipseEdge* are ALWAYS represented in counter-clockwise orientation, even if an clockwise oriented edge is required to build a closed loop. To add a clockwise oriented curve swap start- and

end angles and set the `ccw` flag to `False` and `ezdxf` will export a correct clockwise orientated curve.

type

Path type as `BoundaryPathType.EDGE` enum

path_type_flags

(bit coded flags)

0	default
1	external
16	outermost

see `PolylinePath.path_type_flags`

edges

List of boundary edges of type `LineEdge`, `ArcEdge`, `EllipseEdge` or `SplineEdge`

source_boundary_objects

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

clear() → None

Delete all edges.

add_line(start, end) → LineEdge

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

Parameters

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

add_arc(center, radius=1., start_angle=0., end_angle=360., ccw:bool=True) → ArcEdge

Add an `ArcEdge`.

Adding Clockwise Oriented Arcs:

Clockwise oriented `ArcEdge` objects are sometimes necessary to build closed loops, but the `ArcEdge` objects are always represented in counter-clockwise orientation. To add a clockwise oriented `ArcEdge` you have to swap the start- and end angle and set the `ccw` flag to `False`, e.g. to add a clockwise oriented `ArcEdge` from 180 to 90 degree, add the `ArcEdge` in counter-clockwise orientation with swapped angles:

```
edge_path.add_arc(center, radius, start_angle=90, end_angle=180, ccw=False)
```

Parameters

- **center** – center point of arc, (x, y)-tuple
- **radius** – radius of circle
- **start_angle** – start angle of arc in degrees (`end_angle` for a clockwise oriented arc)
- **end_angle** – end angle of arc in degrees (`start_angle` for a clockwise oriented arc)
- **ccw** – True for counter clockwise False for clockwise orientation

add_ellipse(center, major_axis_vector=(1., 0.), minor_axis_length=1., start_angle=0., end_angle=360., ccw:bool=True) → EllipsePath

Add an `EllipseEdge`.

Adding Clockwise Oriented Ellipses:

Clockwise oriented `EllipseEdge` objects are sometimes necessary to build closed loops, but the `EllipseEdge` objects are always represented in counter-clockwise orientation. To add a clockwise oriented `EllipseEdge` you have to swap the start- and end angle and set the `ccw` flag to `False`, e.g. to add a clockwise oriented `EllipseEdge` from 180 to 90 degree, add the `EllipseEdge` in counter-clockwise orientation with swapped angles:

```
edge_path.add_ellipse(center, major_axis, ratio, start_angle=90, end_
˓→angle=180, ccw=False)
```

Parameters

- **center** – center point of ellipse, (x, y)-tuple
- **major_axis** – vector of major axis as (x, y)-tuple
- **ratio** – ratio of minor axis to major axis as float
- **start_angle** – start angle of ellipse in degrees (`end_angle` for a clockwise oriented ellipse)
- **end_angle** – end angle of ellipse in degrees (`start_angle` for a clockwise oriented ellipse)
- **ccw** – True for counter clockwise False for clockwise orientation

`add_spline(fit_points=None, control_points=None, knot_values=None, weights=None, degree=3, rational=0, periodic=0) → SplinePath`

Add a `SplineEdge`.

Parameters

- **fit_points** – points through which the spline must go, at least 3 fit points are required. list of (x, y)-tuples
- **control_points** – affects the shape of the spline, mandatory and AutoCAD crashes on invalid data. list of (x, y)-tuples
- **knot_values** – (knot vector) mandatory and AutoCAD crashes on invalid data. list of floats; `ezdxf` provides two tool functions to calculate valid knot values: `ezdxf.math.uniform_knot_vector()`, `ezdxf.math.open_uniform_knot_vector()` (default if None)
- **weights** – weight of control point, not mandatory, list of floats.
- **degree** – degree of spline (int)
- **periodic** – 1 for periodic spline, 0 for none periodic spline
- **start_tangent** – start_tangent as 2d vector, optional
- **end_tangent** – end_tangent as 2d vector, optional

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

```
class ezdxf.entities.EdgeType
```

LINE**ARC****ELLIPSE****SPLINE****class ezdxf.entities.LineEdge**

Straight boundary edge.

typeEdge type as *EdgeType.LINE* enum**start**

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

end

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

class ezdxf.entities.ArcEdgeArc as boundary edge in counter-clockwise orientation, see *EdgePath.add_arc()*.**type**Edge type as *EdgeType.ARC* enum**center**

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

radius

Arc radius as float. (read/write)

start_angle

Arc start angle in counter-clockwise orientation in degrees. (read/write)

end_angle

Arc end angle in counter-clockwise orientation in degrees. (read/write)

ccw

True for counter clockwise arc else False. (read/write)

class ezdxf.entities.EllipseEdgeElliptic arc as boundary edge in counter-clockwise orientation, see *EdgePath.add_ellipse()*.**type**Edge type as *EdgeType.ELLIPSE* enum**major_axis_vector**

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

minor_axis_length

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

radius

Ellipse radius as float. (read/write)

start_angle

Ellipse start angle in counter-clockwise orientation in degrees. (read/write)

end_angle

Ellipse end angle in counter-clockwise orientation in degrees. (read/write)

ccw

True for counter clockwise ellipse else False. (read/write)

```
class ezdxf.entities.SplineEdge
    Spline as boundary edge.

    type
        Edge type as EdgeType.SPLINE enum

    degree
        Spline degree as int. (read/write)

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

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

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

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

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

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

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

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

Hatch Pattern Definition Helper Classes

```
class ezdxf.entities.Pattern

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

    add_line(angle: float = 0, base_point: Vertex = (0, 0), offset: Vertex = (0, 0), dash_length_items:
              Iterable[float] = None) → None
        Create a new pattern definition line and add the line to the Pattern.lines attribute.

    clear() → None
        Delete all pattern definition lines.

    scale(factor: float = 1, angle: float = 0) → None
        Scale and rotate pattern.

        Be careful, this changes the base pattern definition, maybe better use Hatch.set_pattern_scale()
        or Hatch.set_pattern_angle().

    Parameters
        • factor – scaling factor
        • angle – rotation angle in degrees

class ezdxf.entities.PatternLine
    Represents a pattern definition line, use factory function Pattern.add_line() to create new pattern definition lines.
```

angle

Line angle in degrees. (read/write)

base_point

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

offset

Offset as (x, y)-tuple. (read/write)

dash_length_items

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

Hatch Gradient Fill Helper Classes

```
class ezdxf.entities.Gradient
```

color1

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

color2

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

one_color

If `one_color` is 1 - the hatch is filled with a smooth transition between `color1` and a specified `tint` of `color1`. (read/write)

rotation

Gradient rotation in degrees. (read/write)

centered

Specifies a symmetrical gradient configuration. If this option is not selected, the gradient fill is shifted up and to the left, creating the illusion of a light source to the left of the object. (read/write)

tint

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

See also:

Tutorial for Hatch Pattern Definition

Image

Add a raster IMAGE ([DXF Reference](#)) to the DXF file, the file itself is not embedded into the DXF file, it is always a separated file. The IMAGE entity is like a block reference, you can use it multiple times to add the image on different locations with different scales and rotations. But therefore you need a also a IMAGEDEF entity, see [*ImageDef*](#). `ezdxf` creates only images in the xy-plan, you can place images in the 3D space too, but then you have to set the `Image.dxf.u_pixel` and the `Image.dxf.v_pixel` vectors by yourself.

Subclass of	<code>ezdxf.entities.DXFGraphic</code>
DXF type	'IMAGE'
Factory function	<code>ezdxf.layouts.BaseLayout.add_image()</code>
Inherited DXF attributes	<i>Common graphical DXF attributes</i>
Required DXF version	DXF R2000 ('AC1015')

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

class ezdxf.entities.**Image****dx_f.insert**

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

dx_f.u_pixel

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

dx_f.v_pixel

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

dx_f.image_size

Image size in pixels as (x, y) tuple

dx_f.image_def_handle

Handle to the image definition entity, see [ImageDef](#)

dx_f.flags

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

dx_f.clipping

Clipping state:

0	clipping off
1	clipping on

dx_f.brightness

Brightness value (0-100; default = 50)

dx_f.contrast

Contrast value (0-100; default = 50)

dx_f.fade

Fade value (0-100; default = 0)

dx_f.clipping_boundary_type

Clipping boundary type:

1	Rectangular
2	Polygonal

dx_f.count_boundary_points

Number of clip boundary vertices, maintained by *ezdxf*.

dxfs.clip_mode

Clip mode (DXF R2010):

0	Outside
1	Inside

boundary_path

A list of vertices as pixel coordinates. Two vertices describe a rectangle, lower left corner is $(-0.5, -0.5)$ and upper right corner is $(\text{ImageSizeX}-0.5, \text{ImageSizeY}-0.5)$, more than two vertices is a polygon as clipping path. All vertices as pixel coordinates. (read/write)

image_def

Returns the associated IMAGEDEF entity, see [ImageDef](#).

reset_boundary_path() → None

Reset boundary path to the default rectangle $[-0.5, -0.5], (\text{ImageSizeX}-0.5, \text{ImageSizeY}-0.5)$.

set_boundary_path(vertices: Iterable[Vertex]) → None

Set boundary path to *vertices*. Two vertices describe a rectangle (lower left and upper right corner), more than two vertices is a polygon as clipping path.

boundary_path_wcs() → List[Vec3]

Returns the boundary/clipping path in WCS coordinates.

New in version 0.14.

Since version 0.16 it's recommended to create the clipping path as [Path](#) object by the [make_path\(\)](#) function:

```
from ezdxf.path import make_path

image = ... # get image entity
clipping_path = make_path(image)
```

transform(m: Matrix44) → Image

Transform IMAGE entity by transformation matrix *m* inplace.

Leader

The LEADER entity ([DXF Reference](#)) represents an arrow, made up of one or more vertices (or spline fit points) and an arrowhead. The label or other content to which the [Leader](#) is attached is stored as a separate entity, and is not part of the [Leader](#) itself.

[Leader](#) shares its styling infrastructure with [Dimension](#).

By default a [Leader](#) without any annotation is created. For creating more fancy leaders and annotations see documentation provided by Autodesk or [Demystifying DXF: LEADER and MULTILEADER implementation notes](#).

Subclass of	<code>ezdxf.entities.DXFGraphic</code>
DXF type	'LEADER'
Factory function	<code>ezdxf.layouts.BaseLayout.add_leader()</code>
Inherited DXF attributes	Common graphical DXF attributes
Required DXF version	DXF R2000 ('AC1015')

class ezdxf.entities.Leader

dx_f.dimstyle

Name of Dimstyle as string.

dx_f.has_arrowhead

0	Disabled
1	Enabled

dx_f.path_type

Leader path type:

0	Straight line segments
1	Spline

dx_f.annotation_type

0	Created with text annotation
1	Created with tolerance annotation
2	Created with block reference annotation
3	Created without any annotation (default)

dx_f.hookline_direction

Hook line direction flag:

0	Hookline (or end of tangent for a splined leader) is the opposite direction from the horizontal vector
1	Hookline (or end of tangent for a splined leader) is the same direction as horizontal vector (see has_hook_line)

dx_f.has_hookline

0	No hookline
1	Has a hookline

dx_f.text_height

Text annotation height in drawing units.

dx_f.text_width

Text annotation width.

dx_f.block_color

Color to use if leader's DIMCLRD = BYBLOCK

dx_f.annotation_handle

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

dx_f.normal_vector

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

.dx_f.horizontal_direction

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

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

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

vertices
List of [Vec3](#) objects, representing the vertices of the leader (3D Point in [WCS](#)).

set_vertices(vertices: Iterable[Vertex])
Set vertices of the leader, vertices is an iterable of (x, y [,z]) tuples or [Vec3](#).

transform(m: Matrix44) → Leader
Transform LEADER entity by transformation matrix *m* inplace.

virtual_entities() → Iterable[Union[Line, Arc]]
Yields ‘virtual’ parts of LEADER as DXF primitives.
This entities are located at the original positions, but are not stored in the entity database, have no handle and are not assigned to any layout.

explode(target_layout: BaseLayout = None) → EntityQuery
Explode parts of LEADER as DXF primitives into target layout, if target layout is `None`, the target layout is the layout of the LEADER.
Returns an [EntityQuery](#) container with all DXF parts.

Parameters `target_layout` – target layout for DXF parts, `None` for same layout as source entity.

New in version 0.14.

Line

LINE ([DXF Reference](#)) entity is a 3D line from `Line.dxf.start` to `Line.dxf.end`.

Subclass of	<code>ezdxf.entities.DXFGraphic</code>
DXF type	'LINE'
Factory function	<code>ezdxf.layouts.BaseLayout.add_line()</code>
Inherited DXF Attributes	<i>Common graphical DXF attributes</i>

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

```
class ezdxf.entities.Line

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

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

dxdf.thickness
    Line thickness in 3D space in direction extrusion, default value is 0. This value should not be confused with the lineweight value.

dxdf.extrusion
    extrusion vector, default value is (0, 0, 1)
```

transform(*m*: Matrix44) → LineTransform the LINE entity by transformation matrix *m* inplace.**translate**(*dx*: float, *dy*: float, *dz*: float) → LineOptimized LINE translation about *dx* in x-axis, *dy* in y-axis and *dz* in z-axis.

LWPolyline

The LWPOLYLINE entity (Lightweight POLYLINE, [DXF Reference](#)) is defined as a single graphic entity, which differs from the old-style [Polyline](#) entity, which is defined as a group of sub-entities. [LWPolyline](#) display faster (in AutoCAD) and consume less disk space, it is a planar element, therefore all points are located in the [OCS](#) as (x, y)-tuples ([LWPolyline.dxf.elevation](#) is the z-axis value).

Changed in version 0.8.9: [LWPolyline](#) stores point data as packed data (array.array).

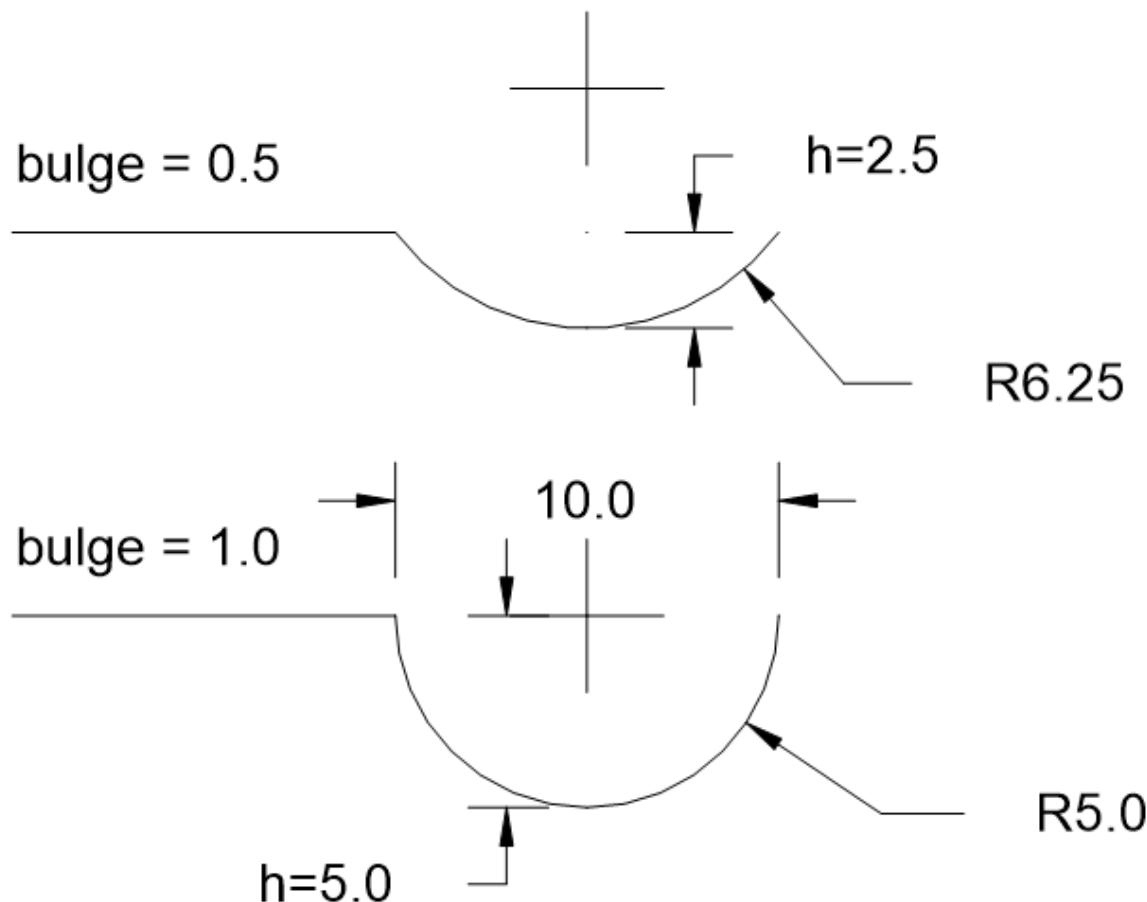
Subclass of	ezdxf.entities.DXFGraphic
DXF type	'LWPOLYLINE'
factory function	add_lwpolyline()
Inherited DXF attributes	Common graphical DXF attributes
Required DXF version	DXF R2000 ('AC1015')

Bulge value

The bulge value is used to create arc shaped line segments for [Polyline](#) and [LWPolyline](#) entities. The arc starts at the vertex which includes the bulge value and ends at the following vertex. The bulge value defines the ratio of the arc sagitta (versine) to half line segment length, a bulge value of 1 defines a semicircle.

The sign of the bulge value defines the side of the bulge:

- positive value (> 0): bulge is right of line (counter clockwise)
- negative value (< 0): bulge is left of line (clockwise)
- 0 = no bulge



Start- and end width

The start width and end width values defines the width in drawing units for the following line segment. To use the default width value for a line segment set value to 0.

Width and bulge values at last point

The width and bulge values of the last point has only a meaning if the polyline is closed, and they apply to the last line segment from the last to the first point.

See also:

[Tutorial for LWPolyline](#) and [Bulge Related Functions](#)

User Defined Point Format Codes

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

```
class ezdxf.entities.LWPolyline
```

dx.f.elevation

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

dx.f.flags

Constants defined in `ezdxf.lldxf.const`:

dx.f.flags	Value	Description
LWPOLYLINE_CLOSED	1	polyline is closed
LWPOLYLINE_PLINEGEN	128	???

dx.f.const_width

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

dx.f.count

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

close(state: bool = True) → None

Get/set closed state of LWPOLYLINE. Compatibility interface to `Polyline`

__len__() → int

Returns count of polyline points.

__getitem__(index: int) → Tuple[float, float, float, float, float]

Returns point at position `index` as (x, y, start_width, end_width, bulge) tuple. start_width, end_width and bulge is 0 if not present, supports extended slicing. Point format is fixed as 'xyseb'.

All coordinates in OCS.

__setitem__(index: int, value: Sequence[float]) → None

Set point at position `index` as (x, y, [start_width, [end_width, [bulge]]]) tuple. If start_width or end_width is 0 or left off the default value is used. If the bulge value is left off, bulge is 0 by default (straight line). Does NOT support extend slicing. Point format is fixed as 'xyseb'.

All coordinates in OCS.

Parameters

- **index** – point index
- **value** – point value as (x, y, [start_width, [end_width, [bulge]]]) tuple

__delitem__(index: int) → None

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

__iter__() → Iterator[Tuple[float, float, float, float, float]]

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

vertices () → Iterator[Tuple[float, float]]

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

vertices_in_wcs () → Iterable[ezdxf.math._vector.Vec3]

Returns iterable of all polyline points as Vec3(x, y, z) in *WCS*.

append (point: Sequence[float], format: str = 'xyseb') → None

Append *point* to polyline, *format* specifies a user defined point format.

All coordinates in *OCS*.

Parameters

- **point** – (x, y, [start_width, [end_width, [bulge]]]) tuple
- **format** – format string, default is 'xyseb', see: [format codes](#)

append_points (points: Iterable[Sequence[float]], format: str = 'xyseb') → None

Append new *points* to polyline, *format* specifies a user defined point format.

All coordinates in *OCS*.

Parameters

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

insert (pos: int, point: Sequence[float], format: str = 'xyseb') → None

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

All coordinates in *OCS*.

Parameters

- **pos** – insert position
- **point** – point data
- **format** – format string, default is 'xyseb', see: [format codes](#)

clear () → None

Remove all points.

get_points (format: str = 'xyseb') → List[Sequence[float]]

Returns all points as list of tuples, *format* specifies a user defined point format.

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

Parameters **format** – format string, default is 'xyseb', see [format codes](#)

set_points (points: Iterable[Sequence[float]], format: str = 'xyseb') → None

Remove all points and append new *points*.

All coordinates in *OCS*.

Parameters

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

points (format: str = 'xyseb') → Iterator[List[Sequence[float]]]

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

All coordinates in *OCS*.

Parameters `format` – format string, see [format codes](#)

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

Transform the LWPOLYLINE entity by transformation matrix *m* inplace.

A non uniform scaling is not supported if the entity contains circular arc segments (bulges).

Parameters `m` – transformation `Matrix44`

Raises `NonUniformScalingError` – for non uniform scaling of entity containing circular arc segments (bulges)

virtual_entities () → Iterable[Union[Line, Arc]]

Yields the graphical representation of LWPOLYLINE as virtual DXF primitives (LINE or ARC).

These virtual entities are located at the original location, but are not stored in the entity database, have no handle and are not assigned to any layout.

explode (*target_layout*: `BaseLayout` = `None`) → EntityQuery

Explode the LWPOLYLINE entity as DXF primitives (LINE or ARC) into the target layout, if the target layout is `None`, the target layout is the layout of the source entity.

Returns an `EntityQuery` container of all DXF primitives.

Parameters `target_layout` – target layout for the DXF primitives, `None` for same layout as the source entity.

MLine

The MLINE entity ([DXF Reference](#)).

Subclass of	<code>ezdxf.entities.DXFGraphic</code>
DXF type	'MLINE'
factory function	<code>add_mline()</code>
Inherited DXF attributes	Common graphical DXF attributes
Required DXF version	DXF R2000 ('AC1015')

class `ezdxf.entities.MLine`

`dxuf.style_name`

`MLineStyle` name stored in `Drawing.mline_styles` dictionary, use `set_style()` to change the MLINESTYLE and update geometry accordingly.

`dxuf.style_handle`

Handle of `MLineStyle`, use `set_style()` to change the MLINESTYLE and update geometry accordingly.

`dxuf.scale_factor`

MLINE scaling factor, use method `set_scale_factor()` to change the scaling factor and update geometry accordingly.

`dxuf.justification`

Justification defines the location of the MLINE in relation to the reference line, use method `set_justification()` to change the justification and update geometry accordingly.

Constants defined in `ezdxf.lldxf.const`:

dx.dxf.justification	Value
MLINE_TOP	0
MLINE_ZERO	1
MLINE_BOTTOM	2
MLINE_RIGHT (alias)	0
MLINE_CENTER (alias)	1
MLINE_LEFT (alias)	2

dx.dxf.flags

Use method `close()` and the properties `start_caps` and `end_caps` to change these flags.

Constants defined in `ezdxf.lldxf.const`:

dx.dxf.flags	Value
MLINE_HAS_VERTEX	1
MLINE_CLOSED	2
MLINE_SUPPRESS_START_CAPS	4
MLINE_SUPPRESS_END_CAPS	8

dx.dxf.start_location

Start location of the reference line. (read only)

dx.dxf.count

Count of MLINE vertices. (read only)

dx.dxf.style_element_count

Count of elements in `MLineStyle` definition. (read only)

dx.dxf.extrusion

Normal vector of the entity plane, but MLINE is not an OCS entity, all vertices of the reference line are WCS! (read only)

vertices

MLINE vertices as `MLineVertex` objects, stored in a regular Python list.

set_style(name: str) → None

Set MLINESTYLE by name and update geometry accordingly. The MLINESTYLE definition must exist.

set_scale_factor(value: float) → None

Set the scale factor and update geometry accordingly.

set_justification(value: int) → None

Set MLINE justification and update geometry accordingly. See `dx.dxf.justification` for valid settings.

close(state: bool = True) → None

Get/set closed state of MLINE and update geometry accordingly. Compatibility interface to `Polyline`.

__len__()

Count of MLINE vertices.

start_location() → Vec3

Returns the start location of the reference line. Callback function for `dx.dxf.start_location`.

get_locations() → List[Vec3]

Returns the vertices of the reference line.

extend(vertices: Iterable[Vec3]) → None

Append multiple vertices to the reference line.

It is possible to work with 3D vertices, but all vertices have to be in the same plane and the normal vector of this plan is stored as extrusion vector in the MLINE entity.

clear() → None

Remove all MLINE vertices.

update_geometry() → None

Regenerate the MLINE geometry based on current settings.

generate_geometry(vertices: List[ezdxf.math._vector.Vec3]) → None

Regenerate the MLINE geometry for new reference line defined by *vertices*.

transform(m: Matrix44) → MLine

Transform MLINE entity by transformation matrix *m* inplace.

virtual_entities() → Iterable[DXFGraphic]

Yields ‘virtual’ parts of MLINE as LINE, ARC and HATCH entities.

This entities are located at the original positions, but are not stored in the entity database, have no handle and are not assigned to any layout.

explode(target_layout: BaseLayout = None) → EntityQuery

Explode parts of MLINE as LINE, ARC and HATCH entities into target layout, if target layout is None, the target layout is the layout of the MLINE.

Returns an *EntityQuery* container with all DXF parts.

Parameters **target_layout** – target layout for DXF parts, None for same layout as source entity.

class ezdxf.entities.**MLineVertex**

location

Reference line vertex location.

line_direction

Reference line direction.

miter_direction

line_params

The line parameterization is a list of float values. The list may contain zero or more items.

The first value (miter-offset) is the distance from the vertex *location* along the *miter_direction* vector to the point where the line element’s path intersects the miter vector.

The next value (line-start-offset) is the distance along the *line_direction* from the miter/line path intersection point to the actual start of the line element.

The next value (dash-length) is the distance from the start of the line element (dash) to the first break (gap) in the line element. The successive values continue to list the start and stop points of the line element in this segment of the mline.

fill_params

The fill parameterization is also a list of float values. Similar to the line parameterization, it describes the parameterization of the fill area for this mline segment. The values are interpreted identically to the line parameters and when taken as a whole for all line elements in the mline segment, they define the boundary of the fill area for the mline segment.

class ezdxf.entities.**MLineStyle**

The *MLineStyle* stores the style properties for the MLINE entity.

dxfs.name

```

dx.f.description
dx.f.flags
dx.f.fill_color
    AutoCAD Color Index (ACI) value of the fill color
dx.f.start_angle
dx.f.end_angle
elements
    MLineStyleElements object
update_all()
    Update all MLINE entities using this MLINESTYLE.
    The update is required if elements were added or removed or the offset of any element was changed.

class ezdxf.entities.mline.MLineStyleElements

elements
    List of MLineStyleElement objects, one for each line element.
MLineStyleElements.__len__()
MLineStyleElements.__getitem__(item)
MLineStyleElements.append(offset: float, color: int = 0, linetype: str = 'BYLAYER') → None
    Append a new line element.

Parameters
    • offset – normal offset from the reference line: if justification is MLINE_ZERO, positive values are above and negative values are below the reference line.
    • color – AutoCAD Color Index (ACI) value
    • linetype – linetype name

class ezdxf.entities.mline.MLineStyleElement
Named tuple to store properties of a line element.

offset
    Normal offset from the reference line: if justification is MLINE_ZERO, positive values are above and negative values are below the reference line.

color
    AutoCAD Color Index (ACI) value

linetype
    Linetype name

```

Mesh

The MESH entity (DXF Reference) is a 3D mesh similar to the `Polyface` entity.

All vertices in `WCS` as (x, y, z) tuples

Changed in version 0.8.9: `Mesh` stores vertices, edges, faces and creases as packed data.

Subclass of	<code>ezdxf.entities.DXFGraphic</code>
DXF type	'MESH'
Factory function	<code>ezdxf.layouts.BaseLayout.add_mesh()</code>
Inherited DXF attributes	<i>Common graphical DXF attributes</i>
Required DXF version	DXF R2000 ('AC1015')

See also:

Tutorial for Mesh and helper classes: `MeshBuilder`, `MeshVertexMerger`

class `ezdxf.entities.Mesh`

dxfs.version
DXF version
0 = off, 1 = on

dxfs.blend_creature
0 for no smoothing else integer greater than 0.

vertices
Vertices as list like `VertexArray`. (read/write)

edges
Edges as list like `TagArray`. (read/write)

faces
Faces as list like `TagList`. (read/write)

creases
Creases as array.array. (read/write)

edit_data() → Iterator[ezdxf.entities.mesh.MeshData]
Context manager various mesh data, returns `MeshData`.
Despite that vertices, edge and faces since *ezdxf* v0.8.9 are accessible as packed data types, the usage of `MeshData` by context manager `edit_data()` is still recommended.

transform(m: Matrix44) → Mesh
Transform the MESH entity by transformation matrix *m* inplace.

MeshData

class `ezdxf.entities.MeshData`

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

faces
A standard Python list with (v1, v2, v3,...) tuples (read/write)
Each face consist of a list of vertex indices (= index in `vertices`).

edges
A standard Python list with (v1, v2) tuples (read/write). These list represents the edges to which the `edge_creature_values` values will be applied. Each edge consist of exact two vertex indices (= index in `vertices`).

edge_creature_values

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

add_face (*vertices: Iterable[Vertex]*) → Sequence[int]

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

add_edge_creature (*v1: int, v2: int, creature: float*)

Add an edge creature value, the edge is defined by the vertex indices *v1* and *v2*. The creature value defines the amount of subdivision that will be applied to this edge. A creature value of the subdivision level prevents the edge from deformation and a value of 0.0 means no protection from subdividing.

optimize (*precision: int = 6*)

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

MPolygon

The MPOLYGON entity is not a core DXF entity and is not supported by every CAD application or DXF library.

The [MPolygon](#) class is very similar to the [Hatch](#) class with small differences in supported DXF attributes and features.

The boundary paths of the MPOLYGON are visible and use the graphical DXF attributes of the main entity like `dxf.color`, `dxf.linetype` and so on. The solid filling is only visible if the attribute `dxf.solid_fill` is 1, the color of the solid fill is defined by `dxf.fill_color` as [AutoCAD Color Index \(ACI\)](#).

MPOLYGON supports `ezdxf.entities.Gradient` settings like HATCH for DXF R2004+. This feature is used by method `MPolygon.set_solid_fill()` to set a solid RGB fill color as linear gradient, this disables pattern fill automatically.

The MPOLYGON does not support associated source path entities, because the MPOLYGON also represents the boundary paths as visible graphical objects.

Hatch patterns are supported, but the hatch style tag is not supported, the default hatch style is `ezdxf.const.HATCH_STYLE_NESTED` and the style flags of the boundary paths are ignored.

Background color for pattern fillings is supported, set background color by property `MPolygon.bgcolor` as RGB tuple.

Note: Background RGB fill color for solid fill and pattern fill is set differently!

Autodesk products do support polyline paths including bulges. An example for edge paths as boundary paths is not available or edge paths are not supported. *Ezdx*f does **not** export MPOLYGON entities including edge paths! The `BoundaryPaths.edge_to_polyline_paths()` method converts all edge paths to simple polyline paths with approximated curves, this conversion has to be done explicit.

See also:

For more information see the `ezdxf.entities.Hatch` documentation.

Subclass of	<code>ezdxf.entities.DXFGraphic</code>
DXF type	'MPOLYGON'
Factory function	<code>ezdxf.layouts.BaseLayout.add_mpolygon()</code>
Inherited DXF attributes	Common graphical DXF attributes
Required DXF version	DXF R2000 ('AC1015')

```
class ezdxf.entities.MPolygon
```

dx_f.pattern_name

Pattern name as string

dx_f.solid_fill

1	solid fill, better use: <code>MPolygon.set_solid_fill()</code>
0	pattern fill, better use: <code>MPolygon.set_pattern_fill()</code>

(search AutoCAD help for more information)

dx_f.pattern_type

0	user
1	predefined
2	custom

dx_f.pattern_angle

Actual pattern angle in degrees (float). Changing this value does not rotate the pattern, use `set_pattern_angle()` for this task.

dx_f.pattern_scale

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

dx_f.pattern_double

1 = double pattern size else 0. (int)

dx_f.elevation

Z value represents the elevation height of the *OCS*. (float)

paths

BoundaryPaths object.

pattern

Pattern object.

gradient

Gradient object.

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

Setup pattern definition by a list of definition lines and a definition line is a 4-tuple (angle, base_point, offset, dash_length_items), the pattern definition should be designed for scaling factor 1 and angle 0.

- angle: line angle in degrees
- base-point: 2-tuple (x, y)
- offset: 2-tuple (dx, dy)
- dash_length_items: list of dash items (item > 0 is a line, item < 0 is a gap and item == 0.0 is a point)

Parameters

- **lines** – list of definition lines
- **factor** – pattern scaling factor

- **angle** – rotation angle in degrees

set_pattern_scale (*scale: float*) → None

Set scaling of pattern definition to *scale*.

Starts always from the original base scaling, `set_pattern_scale(1)` reset the pattern scaling to the original appearance as defined by the pattern designer, but only if the the pattern attribute `dxfs.pattern_scale` represents the actual scaling, it is not possible to recreate the original pattern scaling from the pattern definition itself.

Parameters **scale** – pattern scaling factor

set_pattern_angle (*angle: float*) → None

Set rotation of pattern definition to *angle* in degrees.

Starts always from the original base rotation 0, `set_pattern_angle(0)` reset the pattern rotation to the original appearance as defined by the pattern designer, but only if the the pattern attribute `dxfs.pattern_angle` represents the actual rotation, it is not possible to recreate the original rotation from the pattern definition itself.

Parameters **angle** – rotation angle in degrees

set_solid_fill (*color: int = 7, style: int = 1, rgb: RGB = None*)

Set `MPolygon` to solid fill mode and removes all gradient and pattern fill related data.

Parameters

- **color** – *AutoCAD Color Index (ACI)*, (0 = BYBLOCK; 256 = BYLAYER)
- **style** – hatch style is not supported by `MPolygon`, just for symmetry to HATCH
- **rgb** – true color value as (r, g, b)-tuple - has higher priority than *color*. True color support requires DXF R2004+

set_pattern_fill (*name: str, color: int = 7, angle: float = 0.0, scale: float = 1.0, double: int = 0, style: int = 1, pattern_type: int = 1, definition=None*) → None

Set `Hatch` and `MPolygon` to pattern fill mode. Removes all gradient related data. The pattern definition should be designed for scaling factor 1. Predefined hatch pattern like “ANSI33” are scaled according to the HEADER variable \$MEASUREMENT for ISO measurement (m, cm, ...), or imperial units (in, ft, ...), this replicates the behavior of BricsCAD.

Parameters

- **name** – pattern name as string
- **color** – pattern color as *AutoCAD Color Index (ACI)*
- **angle** – angle of pattern fill in degrees
- **scale** – pattern scaling as float
- **double** – double size flag
- **style** – hatch style (0 = normal; 1 = outer; 2 = ignore)
- **pattern_type** – pattern type (0 = user-defined; 1 = predefined; 2 = custom)
- **definition** – list of definition lines and a definition line is a 4-tuple [angle, base_point, offset, dash_length_items], see `set_pattern_definition()`

set_gradient (*color1: Tuple[int, int, int] = (0, 0, 0), color2: Tuple[int, int, int] = (255, 255, 255), rotation: float = 0.0, centered: float = 0.0, one_color: int = 0, tint: float = 0.0, name: str = 'LINEAR'*) → None

Set `Hatch` and `MPolygon` to gradient fill mode and removes all pattern fill related data. Gradient support requires DXF R2004+. A gradient filled hatch is also a solid filled hatch.

Valid gradient type names are:

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

Parameters

- **color1** – (r, g, b)-tuple for first color, rgb values as int in the range [0, 255]
- **color2** – (r, g, b)-tuple for second color, rgb values as int in the range [0, 255]
- **rotation** – rotation angle in degrees
- **centered** – determines whether the gradient is centered or not
- **one_color** – 1 for gradient from *color1* to tinted *color1*
- **tint** – determines the tinted target *color1* for a one color gradient. (valid range 0.0 to 1.0)
- **name** – name of gradient type, default “LINEAR”

transform(*m*: Matrix44) → MPolygon

Transform entity by transformation matrix *m* inplace.

MText

The MTEXT entity ([DXF Reference](#)) fits a multiline text in a specified width but can extend vertically to an indefinite length. You can format individual words or characters within the [MText](#).

See also:

[Tutorial for MText and MTextEditor](#)

Subclass of	<code>ezdxf.entities.DXFGraphic</code>
DXF type	'MTEXT'
Factory function	<code>ezdxf.layouts.BaseLayout.add_mtext()</code>
Inherited DXF attributes	Common graphical DXF attributes
Required DXF version	DXF R2000 ('AC1015')

class `ezdxf.entities.MText`

`dxfs.insert`

Insertion point (3D Point in [OCS](#))

`dxf.char_height`
Initial text height (float); default=1.0

`dxf.width`
Reference text width (float), forces text wrapping at given width.

`dxf.attachment_point`
Constants defined in `ezdxf.lldxf.const`:

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

`dxf.flow_direction`
Constants defined in `ezdxf.const`:

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

`dxf.style`
Text style (string); default = 'STANDARD'

`dxf.text_direction`
X-axis direction vector in `WCS` (3D Point); default value is (1, 0, 0); if `dxf.rotation` and `dxf.text_direction` are present, `dxf.text_direction` wins.

`dxf.rotation`
Text rotation in degrees (float); default = 0

`dxf.line_spacing_style`
Line spacing style (int), see table below

`dxf.line_spacing_factor`
Percentage of default (3-on-5) line spacing to be applied. Valid values range from 0.25 to 4.00 (float).

Constants defined in `ezdxf.lldxf.const`:

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

`dxf.bg_fill`
Defines the background fill type. (DXF R2007)

MText.dxf.bg_fill	Value	Description
MTEXT_BG_OFF	0	no background color
MTEXT_BG_COLOR	1	use specified color
MTEXT_BG_WINDOW_COLOR	2	use window color (?)
MTEXT_BG_CANVAS_COLOR	3	use canvas background color

dx.dxf.box_fill_scale

Determines how much border there is around the text. (DXF R2007)

Requires: `bg_fill`, `bg_fill_color` else AutoCAD complains

Better use `set_bg_color()`

dx.dxf.bg_fill_color

Background fill color as *AutoCAD Color Index (ACI)* (DXF R2007)

Better use `set_bg_color()`

dx.dxf.bg_fill_true_color

Background fill color as true color value (DXF R2007), also `dx.dxf.bg_fill_color` must be present, else AutoCAD complains.

Better use `set_bg_color()`

dx.dxf.bg_fill_color_name

Background fill color as name string (?) (DXF R2007), also `dx.dxf.bg_fill_color` must be present, else AutoCAD complains.

Better use `set_bg_color()`

dx.dxf.transparency

Transparency of background fill color (DXF R2007), not supported by AutoCAD or BricsCAD.

text

MTEXT content as string (read/write).

Line endings `\n` will be replaced by the MTEXT line endings `\P` at DXF export, but **not** vice versa `\P` by `\n` at DXF file loading.

set_location(insert: Vertex, rotation: float = None, attachment_point: int = None) → MText

Set attributes `dx.dxf.insert`, `dx.dxf.rotation` and `dx.dxf.attachment_point`, `None` for `dx.dxf.rotation` or `dx.dxf.attachment_point` preserves the existing value.

get_rotation() → float

Get text rotation in degrees, independent if it is defined by `dx.dxf.rotation` or `dx.dxf.text_direction`.

set_rotation(angle: float) → ezdxf.entities.mtext.MText

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

get_text_direction() → ezdxf.math._vector.Vec3

Returns the horizontal text direction as `Vec3` object, even if only the text rotation is defined.

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

Set background color as *AutoCAD Color Index (ACI)* value or as name string or as RGB tuple (`r`, `g`, `b`).

Use special color name `canvas`, to set background color to canvas background color.

Use `color = None` to remove the background filling.

Setting only a text border is supported (`color='None'`), but in this case the scaling is always 1.5.

Parameters

- **color** – color as *AutoCAD Color Index (ACI)*, string, RGB tuple or `None`
- **scale** – determines how much border there is around the text, the value is based on the text height, and should be in the range of [1, 5], where 1 fits exact the MText entity.
- **text_frame** – draw a text frame in text color if `True`

__iadd__(text: str) → MText

Append `text` to existing content (`text` attribute).

append(text: str) → MText

Append `text` to existing content (`text` attribute).

plain_text(split=False, fast=True) → Union[List[str], str]

Returns the text content without inline formatting codes.

The “fast” mode is accurate if the DXF content was created by reliable (and newer) CAD applications like AutoCAD or BricsCAD. The “accurate” mode is for some rare cases where the content was created by older CAD applications or unreliable DXF libraries and CAD applications.

Parameters

- **split** – split content text at line breaks if `True` and returns a list of strings without line endings
- **fast** – uses the “fast” mode to extract the plain MTEXT content if `True` or the “accurate” mode if set to `False`

New in version 0.16.6: `fast` argument

all_columns_plain_text(split=False) → Union[List[str], str]

Returns the text content of all columns without inline formatting codes.

Parameters `split` – split content text at line breaks if `True` and returns a list of strings without line endings

New in version 0.17.

all_columns_raw_content() → str

Returns the text content of all columns as a single string including the inline formatting codes.

New in version 0.17.

transform(m: Matrix44) → MText

Transform the MTEXT entity by transformation matrix `m` inplace.

ucs(m: Matrix44) → UCS

Returns the `UCS` of the `MText` entity, defined by the insert location (origin), the text direction or rotation (x-axis) and the extrusion vector (z-axis).

MText Inline Codes

Code	Description
\U	Start underline
\I	Stop underline
\O	Start overline
\o	Stop overline
\K	Start strike-through
\k	Stop strike-through
\P	New paragraph (new line)
\p	Paragraphs properties: indentation, alignment, tabulator stops
\X	Paragraph wrap on the dimension line (only in dimensions)
\Q	Slanting (oblique) text by angle - e.g. \Q30;
\H	Text height - e.g. relative \H3x; absolut \H3;
\W	Text width - e.g. relative \W0.8x; absolut \W0.8;
\T	Tracking, character spacing - e.g. relative \T0.5x; absolut \T2;
\F	Font selection e.g. \Fgdt;o - GDT-tolerance
\S	Stacking, fractions e.g. \SA^ B; space after “^” is required to avoid caret decoding, \SX/Y; \S1#4;
\A	Alignment <ul style="list-style-type: none"> • \A0; = bottom • \A1; = center • \A2; = top
\C	Color change <ul style="list-style-type: none"> • \C1; = red • \C2; = yellow • \C3; = green • \C4; = cyan • \C5; = blue • \C6; = magenta • \C7; = white
\~	Non breaking space
{}	Braces - define the text area influenced by the code, codes and braces can be nested up to 8 levels deep
\	Escape character - e.g. \{ = “{”

Convenient constants defined in MTextEditor:

Constant	Description
UNDERLINE_START	start underline text
UNDERLINE_STOP	stop underline text
OVERSTRIKE_START	start overline
OVERSTRIKE_STOP	stop overline
STRIKE_START	start strike trough
STRIKE_STOP	stop strike trough
GROUP_START	start of group
GROUP_END	end of group
NEW_LINE	start in new line
NBSP	none breaking space

MultiLeader

New in version 0.18.

The MULTILEADER entity ([DXF Reference](#)) represents one or more leaders, made up of one or more vertices (or spline fit points) and an arrowhead. In contrast to the [Leader](#) entity the text- or block content is part of the MULTILEADER entity.

AutoCAD, BricsCAD and maybe other CAD applications do accept 'MLEADER' as type string but they always create entities with 'MULTILEADER' as type string.

Because of the complexity of the MLEADER entity it is recommended to use the [MultiLeaderBuilder](#) to construct the entity.

The visual design is based on an associated [MLeaderStyle](#), but almost all attributes are also stored in the MULTILEADER entity itself.

The attribute [MultiLeader.dxf.property_override_flags](#) should indicate which MLEADERSTYLE attributes are overridden by MULTILEADER attributes, but these flags do not always reflect the state of overridden attributes. The `ezdxf` MULTILEADER renderer uses always the attributes from the MULTILEADER entity and ignores the override flags.

All vertices are WCS coordinates, even those for BLOCK entities which are OCS coordinates in the usual case.

See also:

- [ezdxf.entities.MLeaderStyle](#)
- [ezdxf.render.MultiLeaderBuilder](#)
- [Tutorial for MultiLeader](#)
- [MULTILEADER Internals](#)

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

```
class ezdxf.entities.MultiLeader
```

`dx.dxf.arrow_head_handle`

handle of the arrow head, see also `ezdxf.render.arrows` module, “closed filled” arrow if not set

`dx.dxf.arrow_head_size`

arrow head size in drawing units

`dx.dxf.block_color`

block color as *raw color* value, default is BY_BLOCK_RAW_VALUE

`dx.dxf.block_connection_type`

0	center extents
1	insertion point

`dx.dxf.block_record_handle`

handle to block record of the BLOCK content

`dx.dxf.block_rotation`

BLOCK rotation in radians

`dx.dxf.block_scale_vector`

Vec3 object which stores the scaling factors for the x-, y- and z-axis

`dx.dxf.content_type`

0	none
1	BLOCK
2	MTEXT
3	TOLERANCE

`dx.dxf.dogleg_length`

dogleg length in drawing units

`dx.dxf.has_dogleg`

`dx.dxf.has_landing`

`dx.dxf.has_text_frame`

`dx.dxf.is_annotation`

`dx.dxf.is_text_direction_negative`

`dx.dxf.leader_extend_to_text`

`dx.dxf.leader_line_color`

leader line color as *raw color* value

`dx.dxf.leader_linetype_handle`

handle of the leader linetype, “CONTINUOUS” if not set

`dx.dxf.leader_lineweight`

`dx.dxf.leader_type`

0	invisible
1	straight line leader
2	spline leader

dx_f.property_override_flags

Each bit shows if the MLEADERSTYLE is overridden by the value in the MULTILEADER entity, but this is not always the case for all values, it seems to be save to always use the value from the MULTILEADER entity.

dx_f.scale

overall scaling factor

dx_f.style_handle

handle to the associated MLEADERSTYLE object

dx_f.text_IPE_align

unknown meaning

dx_f.text_alignment_type

unknown meaning - its not the MTEXT attachment point!

dx_f.text_angle_type

0	text angle is equal to last leader line segment angle
1	text is horizontal
2	text angle is equal to last leader line segment angle, but potentially rotated by 180 degrees so the right side is up for readability.

dx_f.text_attachment_direction

defines whether the leaders attach to the left & right of the content BLOCK/MTEXT or attach to the top & bottom:

0	horizontal - left & right of content
1	vertical - top & bottom of content

dx_f.text_attachment_point

MTEXT attachment point

1	top left
2	top center
3	top right

dx_f.text_bottom_attachment_type

9	center
10	overline and center

dx_f.text_color

MTEXT color as *raw color* value

dx_f.text_left_attachment_type

0	top of top MTEXT line
1	middle of top MTEXT line
2	middle of whole MTEXT
3	middle of bottom MTEXT line
4	bottom of bottom MTEXT line
5	bottom of bottom MTEXT line & underline bottom MTEXT line
6	bottom of top MTEXT line & underline top MTEXT line
7	bottom of top MTEXT line
8	bottom of top MTEXT line & underline all MTEXT lines

`dx.dxf.text_right_attachment_type`

0	top of top MTEXT line
1	middle of top MTEXT line
2	middle of whole MTEXT
3	middle of bottom MTEXT line
4	bottom of bottom MTEXT line
5	bottom of bottom MTEXT line & underline bottom MTEXT line
6	bottom of top MTEXT line & underline top MTEXT line
7	bottom of top MTEXT line
8	bottom of top MTEXT line & underline all MTEXT lines

`dx.dxf.text_style_handle`

handle of the MTEXT text style, “Standard” if not set

`dx.dxf.text_top_attachment_type`

9	center
10	overline and center

`dx.dxf.version`

always 2?

`context`

MLeaderContext instance

`arrow_heads`

list of *ArrowHeadData*

`block_attribs`

list of *AttribData*

`virtual_entities()` → Iterable[DXFGraphic]

Yields the graphical representation of MULTILEADER as virtual DXF primitives.

This entities are located at the original location, but are not stored in the entity database, have no handle and are not assigned to any layout.

`explode(target_layout: BaseLayout = None)` → EntityQuery

Explode MULTILEADER as DXF primitives into target layout, if target layout is None, the target layout is the layout of the source entity.

Returns an *EntityQuery* container with all DXF primitives.

Parameters `target_layout` – target layout for the DXF primitives, `None` for same layout as the source entity.

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

Transform the MULTILEADER entity by transformation matrix *m* inplace.

Non uniform scaling is not supported.

Parameters `m` – transformation `Matrix44`

Raises `NonUniformScalingError` – for non uniform scaling

class `ezdxf.entities.MLeaderContext`

leaders

list of `LeaderData` objects

scale

redundant data: `MultiLeader.dxf.scale`

base_point

insert location as `Vec3` of the MTEXT or the BLOCK entity?

char_height

MTEXT char height, already scaled

arrow_head_size

redundant data: `MultiLeader.dxf.arrow_head_size`

landing_gap_size

left_attachment

redundant data: `MultiLeader.dxf.text_left_attachment_type`

right_attachment

redundant data: `MultiLeader.dxf.text_right_attachment_type`

text_align_type

redundant data: `MultiLeader.dxf.text_attachment_point`

attachment_type

BLOCK alignment?

0	content extents
1	insertion point

mtext

instance of `MTextData` if content is MTEXT otherwise `None`

block

instance of `BlockData` if content is BLOCK otherwise `None`

plane_origin

`Vec3`

plane_x_axis

`Vec3`

plane_y_axis

`Vec3`

```
plane_normal_reversed
    the plan normal is x-axis “cross” y-axis (right-hand-rule), this flag indicates to invert this plan normal

top_attachment
    redundant data: MultiLeader.dxf.text_top_attachment_type

bottom_attachment
    redundant data: MultiLeader.dxf.text_bottom_attachment_type

class ezdxf.entities.LeaderData

    lines
        list of LeaderLine

    has_last_leader_line
        unknown meaning

    has_dogleg_vector

    last_leader_point
        WCS point as Vec3

    dogleg_vector
        WCS direction as Vec3

    dogleg_length
        redundant data: MultiLeader.dxf.dogleg_length

    index
        leader index?

    attachment_direction
        redundant data: MultiLeader.dxf.text_attachment_direction

    breaks
        list of break vertices as Vec3 objects

class ezdxf.entities.LeaderLine

    vertices
        list of WCS coordinates as Vec3

    breaks
        mixed list of mixed integer indices and break coordinates or None leader lines without breaks in it

    index
        leader line index?

    color
        leader line color override, ignore override value if BY_BLOCK_RAW_VALUE

class ezdxf.entities.ArrowHeadData

    index
        arrow head index?

    handle
        handle to arrow head block

class ezdxf.entities.AttribData
```

```

handle
    handle to Attdef entity in the BLOCK definition

index
    unknown meaning

width
    text width factor?

text
    Attrib content

class ezdxf.entities.MTextData
    stores the content and attributes of the MTEXT entity

default_content
    content as string

extrusion
    extrusion vector of the MTEXT entity but MTEXT is not an OCS entity!

style_handle
    redundant data: MultiLeader.dxf.text_style_handle

insert
    insert location in WCS coordinates, same as MLeaderContext.base_point?

text_direction
    “horizontal” text direction vector in WCS

rotation
    rotation angle in radians (!) around the extrusion vector, calculated as it were an OCS entity

width
    unscaled column width

defined_height
    unscaled defined column height

line_spacing_factor
    see MText.dxf.line_spacing_factor

line_spacing_style
    see MText.dxf.line_spacing_style

color
    redundant data: MultiLeader.dxf.text_color

alignment
    redundant data: MultiLeader.dxf.text_attachment_point

flow_direction

```

1	horizontal
3	vertical
6	by text style

bg_color
background color as *raw color* value

bg_scale_factor
see *MText.dxf.box_fill_scale*

bg_transparency
background transparency value

use_window_bg_color

has_bg_fill

column_type
unknown meaning - most likely:

0	none
1	static
2	dynamic

use_auto_height

column_width
unscaled column width, redundant data *width*

column_gutter_width
unscaled column gutter width

column_flow_reversed

column_sizes
list of unscaled columns heights for dynamic column with manual heights

use_word_break

class ezdxf.entities.BlockData
stores the attributes for the *Insert* entity

block_record_handle
redundant data: *MultiLeader.dxf.block_record_handle*

extrusion
extrusion vector in WCS

insert
insertion location in WCS as Vec3, same as *MLeaderContext.base_point?*

scale
redundant data: *MultiLeader.dxf.block_scale_vector*

rotation
redundant data: *MultiLeader.dxf.block_rotation*

color
redundant data: *MultiLeader.dxf.block_color*

Point

POINT ([DXF Reference](#)) at location `dx.dxf.location`.

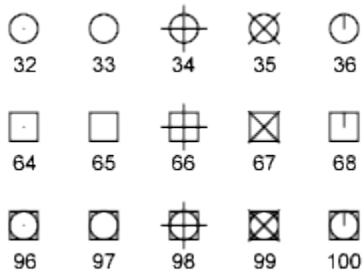
The POINT styling is a global setting, stored as header variable `$PDMODE`, this also means **all** POINT entities in a DXF document have the same styling:

0	center dot (.)
1	none ()
2	cross (+)
3	x-cross (x)
4	tick (‘)

Combined with these bit values

32	circle
64	Square

e.g. circle + square + center dot = $32 + 64 + 0 = 96$



The size of the points is defined by the header variable `$PDSIZE`:

0	5% of draw area height
<0	Specifies a percentage of the viewport size
>0	Specifies an absolute size

Subclass of	<code>ezdxf.entities.DXFGraphic</code>
DXF type	'POINT'
Factory function	<code>ezdxf.layouts.BaseLayout.add_point()</code>
Inherited DXF attributes	<code>Common graphical DXF attributes</code>

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

```
class ezdxf.entities.Point

    dxf.location
        Location of the point (2D/3D Point in WCS)

    dxf.angle
        Angle in degrees of the x-axis for the UCS in effect when POINT was drawn (float); used when PDMODE is nonzero.

    transform(m: Matrix44) → Point
        Transform the POINT entity by transformation matrix m inplace.

    translate(dx: float, dy: float, dz: float) → Point
        Optimized POINT translation about dx in x-axis, dy in y-axis and dz in z-axis.
```

virtual_entities (*pdsiz*: float = 1, *pdmode*: int = 0) → List[DXFGraphic]
Yields point graphic as DXF primitives LINE and CIRCLE entities. The dimensionless point is rendered as zero-length line!

Check for this condition:

```
e.dxftype() == 'LINE' and e.dxf.start.isclose(e.dxf.end)
```

if the rendering engine can't handle zero-length lines.

Parameters

- **pdsiz** – point size in drawing units
- **pdmode** – point styling mode

Polyline

The POLYLINE entity (POLYLINE DXF Reference) is very complex, it's used to build 2D/3D polylines, 3D meshes and 3D polyfaces. For every type exists a different wrapper class but they all have the same dxftype of 'POLYLINE'. Detect POLYLINE type by *Polyline.get_mode()*.

POLYLINE types returned by *Polyline.get_mode()*:

- 'AcDb2dPolyline' for 2D *Polyline*
- 'AcDb3dPolyline' for 3D *Polyline*
- 'AcDbPolygonMesh' for *Polymesh*
- 'AcDbPolyFaceMesh' for *Polyface*

For 2D entities all vertices in *OCS*.

For 3D entities all vertices in *WCS*.

Subclass of	<i>ezdxf.entities.DXFGraphic</i>
DXF type	'POLYLINE'
2D factory function	<i>ezdxf.layouts.BaseLayout.add_polyline2d()</i>
3D factory function	<i>ezdxf.layouts.BaseLayout.add_polyline3d()</i>
Inherited DXF attributes	<i>Common graphical DXF attributes</i>

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

class ezdxf.entities.Polyline

Vertex entities are stored in a standard Python list *Polyline.vertices*. Vertices can be retrieved and deleted by direct access to *Polyline.vertices* attribute:

```
# delete first and second vertex
del polyline.vertices[:2]
```

dxfs.elevation

Elevation point, the X and Y values are always 0, and the Z value is the polyline's elevation (3D Point in *OCS* when 2D, *WCS* when 3D).

dx_f.flagsConstants defined in `ezdxf.1ldxf.const`:

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

dx_f.default_start_width

Default line start width (float); default = 0

dx_f.default_end_width

Default line end width (float); default = 0

dx_f.m_count

Polymesh M vertex count (int); default = 1

dx_f.n_count

Polymesh N vertex count (int); default = 1

dx_f.m_smooth_density

Smooth surface M density (int); default = 0

dx_f.n_smooth_density

Smooth surface N density (int); default = 0

dx_f.smooth_type

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

Constants for `smooth_type` defined in `ezdxf.1ldxf.const`:

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

verticesList of `Vertex` entities.**is_2d_polyline**

True if POLYLINE is a 2D polyline.

is_3d_polyline

True if POLYLINE is a 3D polyline.

is_polygon_mesh

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

is_poly_face_mesh

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

is_closed

True if POLYLINE is closed.

is_m_closed

True if POLYLINE (as *Polymesh*) is closed in m direction.

is_n_closed

True if POLYLINE (as *Polymesh*) is closed in n direction.

has_arc

Returns True if 2D POLYLINE has an arc segment.

has_width

Returns True if 2D POLYLINE has default width values or any segment with width attributes.

New in version 0.14.

get_mode () → str

Returns POLYLINE type as string:

- ‘AcDb2dPolyline’
- ‘AcDb3dPolyline’
- ‘AcDbPolygonMesh’
- ‘AcDbPolyFaceMesh’

m_close (status=True) → None

Close POLYMESH in m direction if *status* is True (also closes POLYLINE), clears closed state if *status* is False.

n_close (status=True) → None

Close POLYMESH in n direction if *status* is True, clears closed state if *status* is False.

close (m_close=True, n_close=False) → None

Set closed state of POLYMESH and POLYLINE in m direction and n direction. True set closed flag, False clears closed flag.

__len__ () → int

Returns count of *Vertex* entities.

__getitem__ (pos) → ezdxf.entities.polyline.DXFVertex

Get *Vertex* entity at position *pos*, supports list slicing.

points () → Iterator[ezdxf.math._vector.Vec3]

Returns iterable of all polyline vertices as (x, y, z) tuples, not as *Vertex* objects.

append_vertex (point: Vertex, dxfattribs=None) → None

Append a single *Vertex* entity at location *point*.

Parameters

- **point** – as (x, y[, z]) tuple
- **dxfattribs** – dict of DXF attributes for *Vertex* class

append_vertices (points: Iterable[Vertex], dxfattribs=None) → None

Append multiple *Vertex* entities at location *points*.

Parameters

- **points** – iterable of (x, y[, z]) tuples
- **dxfattribs** – dict of DXF attributes for the VERTEX objects

append_formatted_vertices(*points*: Iterable[Vertex], *format*: str = 'xy', *dxfattribs*=None) → None

Append multiple *Vertex* entities at location *points*.

Parameters

- **points** – iterable of (x, y, [start_width, [end_width, [bulge]]]) tuple
- **format** – format string, default is 'xy', see: *User Defined Point Format Codes*
- **dxfattribs** – dict of DXF attributes for the VERTEX objects

insert_vertices(*pos*: int, *points*: Iterable[Vertex], *dxfattribs*=None) → None

Insert vertices *points* into *Polyline.vertices* list at insertion location *pos*.

Parameters

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

transform(*m*: Matrix44) → Polyline

Transform the POLYLINE entity by transformation matrix *m* inplace.

A non uniform scaling is not supported if a 2D POLYLINE contains circular arc segments (bulges).

Parameters *m* – transformation *Matrix44*

Raises NonUniformScalingError – for non uniform scaling of 2D POLYLINE containing circular arc segments (bulges)

virtual_entities() → Iterable[Union[Line, Arc, Face3d]]

Yields the graphical representation of POLYLINE as virtual DXF primitives (LINE, ARC or 3DFACE).

These virtual entities are located at the original location, but are not stored in the entity database, have no handle and are not assigned to any layout.

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

Explode the POLYLINE entity as DXF primitives (LINE, ARC or 3DFACE) into the target layout, if the target layout is None, the target layout is the layout of the POLYLINE entity.

Returns an *EntityQuery* container of all DXF primitives.

Parameters *target_layout* – target layout for DXF primitives, None for same layout as source entity.

Vertex

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

Subclass of	<i>ezdxf.entities.DXFGraphic</i>
DXF type	'VERTEX'
Factory function	<i>Polyline.append_vertex()</i>
Factory function	<i>Polyline.extend()</i>
Factory function	<i>Polyline.insert_vertices()</i>
Inherited DXF Attributes	<i>Common graphical DXF attributes</i>

```
class ezdxf.entities.Vertex

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

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

    dxf.end_width
        Line segment end width (float); default = 0

    dxf.bulge
        Bulge value (float); default = 0.

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

    dxf.flags
        Constants defined in ezdxf.lldxf.const:
```

Vertex.dxf.flags	Value	Description
VTX_EXTRA_VERTEX1_CREATED		vertex created by curve-fitting
VTX_CURVE_FIT_TANGENT		curve-fit tangent defined for this vertex. A curve-fit tangent direction of 0 may be omitted from the DXF output, but is significant if this bit is set.
VTX_SPLINE_VERTEX8_CREATED		vertex created by spline-fitting
VTX_SPLINE_FRAME_CONTROL_POINT		control point
VTX_3D_POLYLINE_VERTEX		polyline vertex
VTX_3D_POLYGON_MESH_VERTEX		polygon mesh
VTX_3D_POLYFACE_MESH_VERTEX		mesh vertex

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

    dxf.vtx1
        Index of 1st vertex, if used as face (feature for experts)

    dxf.vtx2
        Index of 2nd vertex, if used as face (feature for experts)

    dxf.vtx3
        Index of 3rd vertex, if used as face (feature for experts)

    dxf.vtx4
        Index of 4th vertex, if used as face (feature for experts)

    is_2d_polyline_vertex
    is_3d_polyline_vertex
    is_polygon_mesh_vertex
    is_poly_face_mesh_vertex
    is_face_record

    format(format='xyz') → Sequence
        Return formatted vertex components as tuple.

        Format codes:
            • “x” = x-coordinate
```

- “y” = y-coordinate
- “z” = z-coordinate
- “s” = start width
- “e” = end width
- “b” = bulge value
- “v” = (x, y, z) as tuple

Args: format: format string, default is “xyz”

New in version 0.14.

Polymesh

Subclass of	<code>ezdxf.entities.Polyline</code>
DXF type	'POLYLINE'
Factory function	<code>ezdxf.layouts.BaseLayout.add_polymesh()</code>
Inherited DXF Attributes	<i>Common graphical DXF attributes</i>

`class ezdxf.entities.Polymesh`

A polymesh is a grid of m_count x n_count vertices, every vertex has its own (x, y, z) location. The `Polymesh` is a subclass of `Polyline`, DXF type is also 'POLYLINE' but `get_mode()` returns 'AcDbPolygonMesh'.

get_mesh_vertex (`pos: Tuple[int, int]`) → `ezdxf.entities.polyline.DXFVertex`
Get location of a single mesh vertex.

Parameters `pos` – 0-based (row, col) tuple, position of mesh vertex

set_mesh_vertex (`pos: Tuple[int, int], point: Vertex, dxfattribs=None`)
Set location and DXF attributes of a single mesh vertex.

Parameters

- `pos` – 0-based (row, col)-tuple, position of mesh vertex
- `point` – (x, y, z)-tuple, new 3D coordinates of the mesh vertex
- `dxfattribs` – dict of DXF attributes

get_mesh_vertex_cache () → `ezdxf.entities.polyline.MeshVertexCache`

Get a `MeshVertexCache` object for this POLYMESH. The caching object provides fast access to the location attribute of mesh vertices.

MeshVertexCache

`class ezdxf.entities.MeshVertexCache`

Cache mesh vertices in a dict, keys are 0-based (row, col) tuples.

Set vertex location: `cache[row, col] = (x, y, z)`

Get vertex location: `x, y, z = cache[row, col]`

`vertices`

Dict of mesh vertices, keys are 0-based (row, col) tuples.

__getitem__ (*pos: Tuple[int, int]*) → *Vertex*
Get mesh vertex location as (x, y, z)-tuple.

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

__setitem__ (*pos: Tuple[int, int], location: Vertex*) → *None*
Get mesh vertex location as (x, y, z)-tuple.

Parameters

- **pos** – 0-based (row, col)-tuple.
- **location** – (x, y, z)-tuple

Polyface

Subclass of	<i>ezdxf.entities.Polyline</i>
DXF type	'POLYLINE'
Factory function	<i>ezdxf.layouts.BaseLayout.add_polyface()</i>
Inherited DXF Attributes	<i>Common graphical DXF attributes</i>

See also:

Tutorial for Polyface

class *ezdxf.entities.Polyface*

A polyface consist of multiple location independent 3D areas called faces. The *Polyface* is a subclass of *Polyline*, DXF type is also 'POLYLINE' but *get_mode()* returns 'AcDbPolyFaceMesh'.

append_face (*face: FaceType, dxfattribs=None*) → *None*
Append a single face. A *face* is a list of (x, y, z) tuples.

Parameters

- **face** – List[(x, y, z) tuples]
- **dxfattribs** – dict of DXF attributes for VERTEX objects

append_faces (*faces: Iterable[FaceType], dxfattribs=None*) → *None*
Append multiple *faces*. *faces* is a list of single faces and a single face is a list of (x, y, z) tuples.

Parameters

- **faces** – list of List[(x, y, z) tuples]
- **dxfattribs** – dict of DXF attributes for teh VERTEX objects

faces() → *Iterable[List[Vertex]]*
Iterable of all faces, a face is a tuple of vertices.

Returns [vertex, vertex, vertex, [vertex,] face_record]

Return type list

optimize (*precision: int = 6*) → *None*
Rebuilds *Polyface* including vertex optimization by merging vertices with nearly same vertex locations.

Parameters **precision** – floating point precision for determining identical vertex locations

Ray

RAY entity ([DXF Reference](#)) starts at Ray.dxf.point and continues to infinity (construction line).

Subclass of	<code>ezdxf.entities.XLine</code>
DXF type	'RAY'
Factory function	<code>ezdxf.layouts.BaseLayout.add_ray()</code>
Inherited DXF attributes	Common graphical DXF attributes
Required DXF version	DXF R2000 ('AC1015')

```
class ezdxf.entities.Ray
```

`dx.dxf.start`

Start point as (3D Point in [WCS](#))

`dx.dxf.unit_vector`

Unit direction vector as (3D Point in [WCS](#))

`transform(m: Matrix44) → Ray`

Transform the XLINE/RAY entity by transformation matrix *m* inplace.

`translate(dx: float, dy: float, dz: float) → Ray`

Optimized XLINE/RAY translation about *dx* in x-axis, *dy* in y-axis and *dz* in z-axis.

Region

REGION ([DXF Reference](#)) created by an ACIS based geometry kernel provided by the Spatial Corp.

See also:

*Ezdx*f will never create or interpret ACIS data, for more information see the FAQ: [How to add/edit ACIS based entities like 3DSOLID, REGION or SURFACE?](#)

Subclass of	<code>ezdxf.entities.Body</code>
DXF type	'REGION'
Factory function	<code>ezdxf.layouts.BaseLayout.add_region()</code>
Inherited DXF attributes	Common graphical DXF attributes
Required DXF version	DXF R2000 ('AC1015')

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

```
class ezdxf.entities.Region
```

Same attributes and methods as parent class `Body`.

Shape

SHAPES ([DXF Reference](#)) are objects that are used like block references, each SHAPE reference can be scaled and rotated individually. The SHAPE definitions are stored in external shape files (*.SHX), and *ezdx*f can not create this shape files.

Subclass of	<code>ezdxf.entities.DXFGraphic</code>
DXF type	'SHAPE'
Factory function	<code>ezdxf.layouts.BaseLayout.add_shape()</code>
Inherited DXF attributes	<i>Common graphical DXF attributes</i>

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

`class ezdxf.entities.Shape`

```
dxfl.insert
    Insertion location as (2D/3D Point in WCS)

dxfl.name
    Shape name (str)

dxfl.size
    Shape size (float)

dxfl.rotation
    Rotation angle in degrees; default value is 0

dxfl.xscale
    Relative X scale factor (float); default value is 1

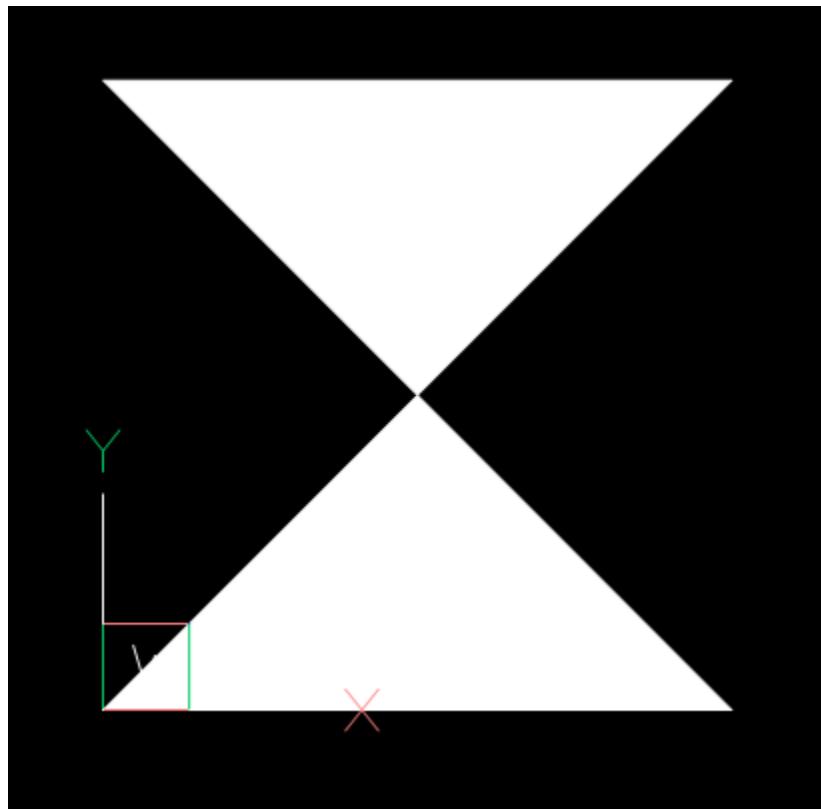
dxfl.oblique
    Oblique angle in degrees (float); default value is 0

transform(m: Matrix44) → Shape
    Transform the SHAPE entity by transformation matrix m inplace.
```

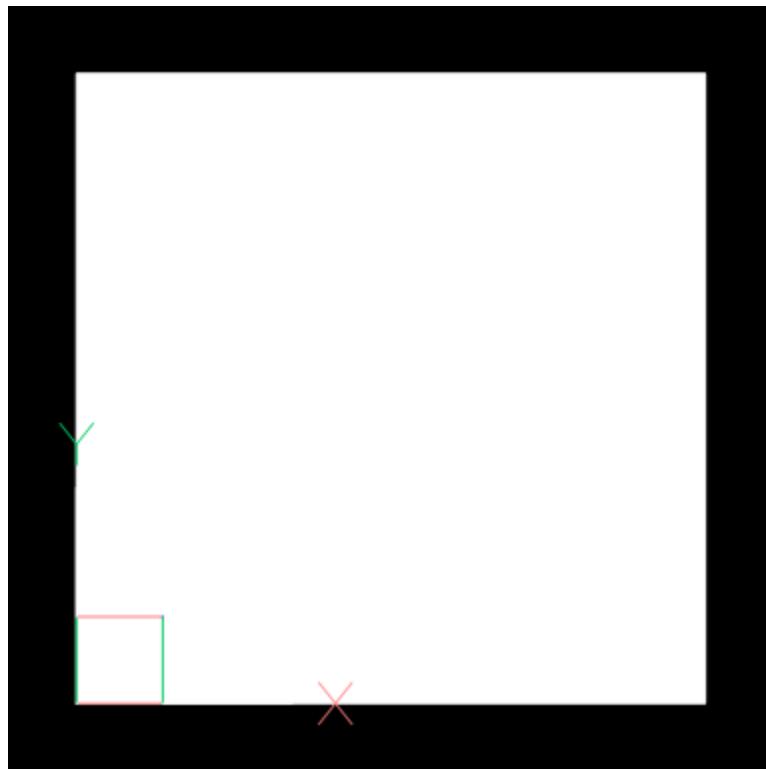
Solid

SOLID ([DXF Reference](#)) is a filled triangle or quadrilateral. Access vertices by name (`entity.dxfl.vtx0 = (1.7, 2.3)`) or by index (`entity[0] = (1.7, 2.3)`). If only 3 vertices are provided the last (3rd) vertex will be repeated in the DXF file.

The SOLID entity stores the vertices in an unusual way, the last two vertices are reversed. The coordinates `[(0, 0), (1, 0), (1, 1), (0, 1)]` do not create a square as you would expect:



Reverse the last two vertices to get the *expected* square: `[(0, 0), (1, 0), (0, 1), (1, 1)]`



Note: The `Solid.vertices()` and `Solid.wcs_vertices()` methods return the vertices in the *expected*

(reversed) order.

Subclass of	<code>ezdxf.entities.DXFGraphic</code>
DXF type	'SOLID'
Factory function	<code>ezdxf.layouts.BaseLayout.add_solid()</code>
Inherited DXF attributes	<i>Common graphical DXF attributes</i>

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

`class ezdxf.entities.Solid`

`dxfs.vtx0`

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

`dxfs.vtx1`

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

`dxfs.vtx2`

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

`dxfs.vtx3`

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

`transform(m: Matrix44) → Solid`

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

`vertices(close: bool=False) → List[Vec3]`

Returns OCS vertices in correct order, if argument *close* is `True`, last vertex == first vertex. Does **not** return duplicated last vertex if represents a triangle.

New in version 0.15.

`wcs_vertices(close: bool=False) → List[Vec3]`

Returns WCS vertices in correct order, if argument *close* is `True`, last vertex == first vertex. Does **not** return duplicated last vertex if represents a triangle.

New in version 0.15.

Spline

SPLINE curve ([DXF Reference](#)), all coordinates have to be 3D coordinates even the spline is only a 2D planar curve.

The spline curve is defined by control points, knot values and weights. The control points establish the spline, the various types of knot vector determines the shape of the curve and the weights of rational splines define how strong a control point influences the shape.

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

New in version 0.16: The function `ezdxf.math.fit_points_to_cad_cv()` calculates control vertices from given fit points. This control vertices define a cubic B-spline which matches visually the SPLINE entities created by BricsCAD and AutoCAD from fit points.

See also:

- Wikipedia article about B_splines
- Department of Computer Science and Technology at the Cambridge University
- *Tutorial for Spline*

Subclass of	<code>ezdxf.entities.DXFGraphic</code>
DXF type	'SPLINE'
Factory function	see table below
Inherited DXF attributes	<i>Common graphical DXF attributes</i>
Required DXF version	DXF R2000 ('AC1015')

Factory Functions

Basic spline entity	<code>add_spline()</code>
Spline control frame from fit points	<code>add_spline_control_frame()</code>
Open uniform spline	<code>add_open_spline()</code>
Closed uniform spline	<code>add_closed_spline()</code>
Open rational uniform spline	<code>add_rational_spline()</code>
Closed rational uniform spline	<code>add_closed_rational_spline()</code>

class `ezdxf.entities.Spline`

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

dx.f.degree

Degree of the spline curve (int).

dx.f.flags

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

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

dx.f.n_knots

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

dx.f.n_fit_points

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

dx.f.n_control_points

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

dx.f.knot_tolerance

Knot tolerance (float); default = $1e-10$

dx.f.fit_tolerance

Fit tolerance (float); default = $1e-10$

dx.f.control_point_tolerance

Control point tolerance (float); default = $1e-10$

dxdf.start_tangent
Start tangent vector as (3D vector in *WCS*)

dxdf.end_tangent
End tangent vector as (3D vector in *WCS*)

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

control_points
VertexArray of control points in *WCS*.

fit_points
VertexArray of fit points in *WCS*.

knots
Knot values as `array.array('d')`.

weights
Control point weights as `array.array('d')`.

control_point_count() → int
Count of control points.

fit_point_count() → int
Count of fit points.

knot_count() → int
Count of knot values.

construction_tool() → BSpline
Returns the construction tool `ezdxf.math.BSpline`.

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

flattening(distance: float, segments: int = 4) → Iterable[Vec3]
Adaptive recursive flattening. The argument *segments* is the minimum count of approximation segments between two knots, if the distance from the center of the approximation segment to the curve is bigger than *distance* the segment will be subdivided.

Parameters

- **distance** – maximum distance from the projected curve point onto the segment chord.
- **segments** – minimum segment count between two knots

New in version 0.15.

set_open_uniform(control_points: Sequence[Vertex], degree: int = 3) → None
Open B-spline with uniform knot vector, start and end at your first and last control points.

set_uniform(control_points: Sequence[Vertex], degree: int = 3) → None
B-spline with uniform knot vector, does NOT start and end at your first and last control points.

set_closed(control_points: Sequence[Vertex], degree=3) → None
Closed B-spline with uniform knot vector, start and end at your first control point.

set_open_rational(control_points: Sequence[Vertex], weights: Sequence[float], degree: int = 3)
→ None
Open rational B-spline with uniform knot vector, start and end at your first and last control points, and has additional control possibilities by weighting each control point.

set_uniform_rational (*control_points*: Sequence[Vertex], *weights*: Sequence[float], *degree*: int = 3) → None
Rational B-spline with uniform knot vector, does NOT start and end at your first and last control points, and has additional control possibilities by weighting each control point.

set_closed_rational (*control_points*: Sequence[Vertex], *weights*: Sequence[float], *degree*: int = 3) → None
Closed rational B-spline with uniform knot vector, start and end at your first control point, and has additional control possibilities by weighting each control point.

transform (*m*: Matrix44) → Spline
Transform the SPLINE entity by transformation matrix *m* inplace.

classmethod from_arc (*entity*: DXFGraphic) → Spline
Create a new SPLINE entity from a CIRCLE, ARC or ELLIPSE entity.
The new SPLINE entity has no owner, no handle, is not stored in the entity database nor assigned to any layout!

Surface

SURFACE ([DXF Reference](#)) created by an ACIS based geometry kernel provided by the Spatial Corp.

See also:

*Ezdx*f will never create or interpret ACIS data, for more information see the FAQ: [How to add/edit ACIS based entities like 3DSOLID, REGION or SURFACE?](#)

Subclass of	<code>ezdxf.entities.Body</code>
DXF type	'SURFACE'
Factory function	<code>ezdxf.layouts.BaseLayout.add_surface()</code>
Inherited DXF attributes	Common graphical DXF attributes
Required DXF version	DXF R2000 ('AC1015')

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

class ezdxf.entities.Surface

Same attributes and methods as parent class `Body`.

dx.f.u_count

Number of U isolines.

dx.f.v_count

Number of V2 isolines.

ExtrudedSurface

([DXF Reference](#))

Subclass of	<code>ezdxf.entities.Surface</code>
DXF type	'EXTRUDED_SURFACE'
Factory function	<code>ezdxf.layouts.BaseLayout.add_extruded_surface()</code>
Inherited DXF attributes	Common graphical DXF attributes
Required DXF version	DXF R2007 ('AC1021')

```
class ezdxf.entities.ExtrudedSurface
    Same attributes and methods as parent class Surface.
    dxflist.class_id
    dxflist.sweep_vector
    dxflist.draft_angle
    dxflist.draft_start_distance
    dxflist.draft_end_distance
    dxflist.twist_angle
    dxflist.scale_factor
    dxflist.align_angle
    dxflist.solid
    dxflist.sweep_alignment_flags
```

0	No alignment
1	Align sweep entity to path
2	Translate sweep entity to path
3	Translate path to sweep entity

```
dxflist.align_start
dxflist.bank
dxflist.base_point_set
dxflist.sweep_entity_transform_computed
dxflist.path_entity_transform_computed
dxflist.reference_vector_for_controlling_twist
transformation_matrix_extruded_entity
    type: Matrix44
sweep_entity_transformation_matrix
    type: Matrix44
path_entity_transformation_matrix
    type: Matrix44
```

LoftedSurface

(DXF Reference)

Subclass of	<i>ezdxf.entities.Surface</i>
DXF type	'LOFTEDSURFACE'
Factory function	<i>ezdxf.layouts.BaseLayout.add_lofted_surface()</i>
Inherited DXF attributes	<i>Common graphical DXF attributes</i>
Required DXF version	DXF R2007 ('AC1021')

```
class ezdxf.entities.LoftedSurface
    Same attributes and methods as parent class Surface.
        dxf.plane_normal_lofting_type
        dxf.start_draft_angle
        dxf.end_draft_angle
        dxf.start_draft_magnitude
        dxf.end_draft_magnitude
        dxf.arc_length_parameterization
        dxf.no_twist
        dxf.align_direction
        dxf.simple_surfaces
        dxf.closed_surfaces
        dxf.solid
        dxf.ruled_surface
        dxf.virtual_guide
set_transformation_matrix_lofted_entity
    type: Matrix44
```

RevolvedSurface

(DXF Reference)

Subclass of	<i>ezdxf.entities.Surface</i>
DXF type	'REVOLVEDSURFACE'
Factory function	<i>ezdxf.layouts.BaseLayout.add_revolved_surface()</i>
Inherited DXF attributes	<i>Common graphical DXF attributes</i>
Required DXF version	DXF R2007 ('AC1021')

```
class ezdxf.entities.RevolvedSurface
    Same attributes and methods as parent class Surface.
        dxf.class_id
        dxf.axis_point
        dxf.axis_vector
        dxf.revolve_angle
        dxf.start_angle
        dxf.draft_angle
        dxf.start_draft_distance
        dxf.end_draft_distance
        dxf.twist_angle
        dxf.solid
```

```
dxf.close_to_axis  
transformation_matrix_revolved_entity  
type: Matrix44
```

SweptSurface

(DXF Reference)

Subclass of	<i>ezdxf.entities.Surface</i>
DXF type	'SWEPTSURFACE'
Factory function	<i>ezdxf.layouts.BaseLayout.add_swept_surface()</i>
Inherited DXF attributes	<i>Common graphical DXF attributes</i>
Required DXF version	DXF R2007 ('AC1021')

```
class ezdxf.entities.SweptSurface  
Same attributes and methods as parent class Surface.  
  
dxf.swept_entity_id  
dxf.path_entity_id  
dxf.draft_angle  
draft_start_distance  
dxf.draft_end_distance  
dxf.twist_angle  
dxf.scale_factor  
dxf.align_angle  
dxf.solid  
dxf.sweep_alignment  
dxf.align_start  
dxf.bank  
dxf.base_point_set  
dxf.sweep_entity_transform_computed  
dxf.path_entity_transform_computed  
dxf.reference_vector_for_controlling_twist  
transformation_matrix_sweep_entity  
type: Matrix44  
transformation_matrix_path_entity()  
type: Matrix44  
sweep_entity_transformation_matrix()  
type: Matrix44  
path_entity_transformation_matrix()  
type: Matrix44
```

Text

The single line TEXT entity ([DXF Reference](#)). The `style` attribute stores the associated `Textstyle` entity as string, which defines the basic font properties. The text size is stored as cap height in the `height` attribute in drawing units. Text alignments are defined as enums of type `ezdxf.enums.TextEntityAlignment`.

See also:

See the documentation for the `Textstyle` class to understand the limitations of text representation in the DXF format.

[Tutorial for Text](#)

Subclass of	<code>ezdxf.entities.DXFGraphic</code>
DXF type	'TEXT'
Factory function	<code>ezdxf.layouts.BaseLayout.add_text()</code>
Inherited DXF attributes	Common graphical DXF attributes

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

```
class eздxf.entities.Text

    dxף.text
        Text content as string.

    dxף.insert
        First alignment point of text (2D/3D Point in OCS), relevant for the adjustments LEFT, ALIGNED and FIT.

    dxף.align_point
        The main alignment point of text (2D/3D Point in OCS), if the alignment is anything else than LEFT, or the second alignment point for the ALIGNED and FIT alignments.

    dxף.height
        Text height in drawing units as float value, the default value is 1.

    dxף.rotation
        Text rotation in degrees as float value, the default value is 0.

    dxף.oblique
        Text oblique angle (slanting) in degrees as float value, the default value is 0 (straight vertical text).

    dxף.style
        Textstyle name as case insensitive string, the default value is "Standard"

    dxף.width
        Width scale factor as float value, the default value is 1.

    dxף.halign
        Horizontal alignment flag as int value, use the set_placement() and get_align_enum() methods to handle text alignment, the default value is 0.
```

0	Left
2	Right
3	Aligned (if vertical alignment = 0)
4	Middle (if vertical alignment = 0)
5	Fit (if vertical alignment = 0)

dx.dxf.valign

Vertical alignment flag as int value, use the `set_placement()` and `get_align_enum()` methods to handle text alignment, the default value is 0.

0	Baseline
1	Bottom
2	Middle
3	Top

dx.dxf.text_generation_flag

Text generation flags as int value, use the `is_backward` and `is_upside_down` attributes to handle this flags.

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

set_placement (p1: Vertex, p2:Vertex=None, align: TextEntityAlignment=None)

Set text alignment and location.

The alignments ALIGNED and FIT are special, they require a second alignment point, the text is aligned on the virtual line between these two points and sits vertically at the base line.

- ALIGNED: Text is stretched or compressed to fit exactly between `p1` and `p2` and the text height is also adjusted to preserve height/width ratio.
- FIT: Text is stretched or compressed to fit exactly between `p1` and `p2` but only the text width is adjusted, the text height is fixed by the `dx.dxf.height` attribute.
- MIDDLE: also a special adjustment, centered text like MIDDLE_CENTER, but vertically centred at the total height of the text.

Parameters

- `p1` – first alignment point as (x, y[, z])
- `p2` – second alignment point as (x, y[, z]), required for ALIGNED and FIT else ignored
- `align` – new alignment as enum `TextEntityAlignment`, None to preserve the existing alignment.

set_pos (p1: Vertex, p2:Vertex=None, align: str=None)

Set text alignment and location. (deprecated)

The alignments “ALIGNED” and “FIT” are special, they require a second alignment point, the text is aligned on the virtual line between these two points and sits vertically at the base line.

- “ALIGNED”: Text is stretched or compressed to fit exactly between `p1` and `p2` and the text height is also adjusted to preserve height/width ratio.
- “FIT”: Text is stretched or compressed to fit exactly between `p1` and `p2` but only the text width is adjusted, the text height is fixed by the `dx.dxf.height` attribute.

- “MIDDLE”: also a special adjustment, centered text like “MIDDLE_CENTER”, but vertically centred at the total height of the text.

Warning: Will be removed in v1.0.0, use `set_placement()`

Parameters

- **p1** – first alignment point as (x, y[, z])
- **p2** – second alignment point as (x, y[, z]), required for “ALIGNED” and “FIT” else ignored
- **align** – new alignment as string or None to preserve the existing alignment.

get_placement() → Tuple[TextEntityAlignment, Vec3, Optional[Vec3]]

Returns a tuple (`align`, `p1`, `p2`), `align` is the alignment enum `TextEntityAlignment`, `p1` is the alignment point, `p2` is only relevant if `align` is `ALIGNED` or `FIT`, otherwise it is `None`.

get_pos() → Tuple[str, Vec3, Optional[Vec3]]

Returns a tuple (`align`, `p1`, `p2`), `align` is the alignment method, `p1` is the alignment point, `p2` is only relevant if `align` is “ALIGNED” or “FIT”, otherwise it is `None` (deprecated).

Warning: Will be removed in v1.0.0, use `get_placement()`

get_align_enum() → ezdxf.enums.TextEntityAlignment

Returns the current text alignment as `TextEntityAlignment`, see also `set_placement()`.

get_align() → str

Returns the current text alignment as string (deprecated).

Warning: Will be removed in v1.0.0, use `get_align_enum()`

set_align_enum(`align = TextEntityAlignment.LEFT`) → Text

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

Parameters align – `TextEntityAlignment`

set_align(`align: str = "LEFT"`) → Text

Set the text alignment as string (deprecated)

Warning: Will be removed in v1.0.0, use `set_align_enum()`

transform(`m: Matrix44`) → Text

Transform the TEXT entity by transformation matrix `m` inplace.

translate(`dx: float, dy: float, dz: float`) → Text

Optimized TEXT/ATTRIB/ATTDEF translation about `dx` in x-axis, `dy` in y-axis and `dz` in z-axis, returns `self`.

plain_text() → str

Returns text content without formatting codes.

font_name() → str

Returns the font name of the associated *Textstyle*.

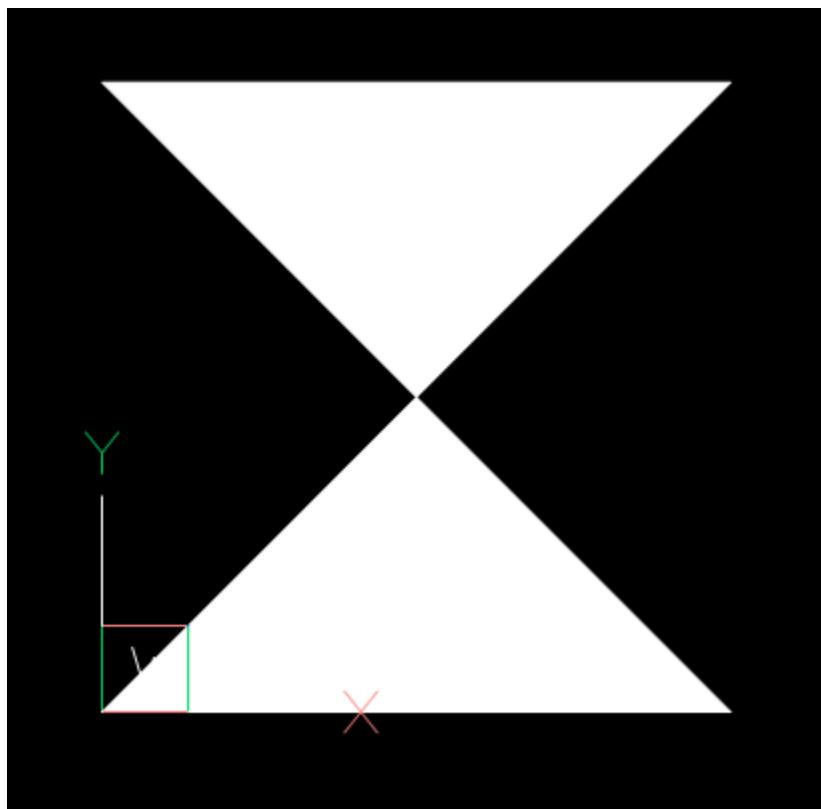
fit_length() → float

Returns the text length for alignments `TextEntityAlignment.FIT` and `TextEntityAlignment.ALIGNED`, defined by the distance from the insertion point to the align point or 0 for all other alignments.

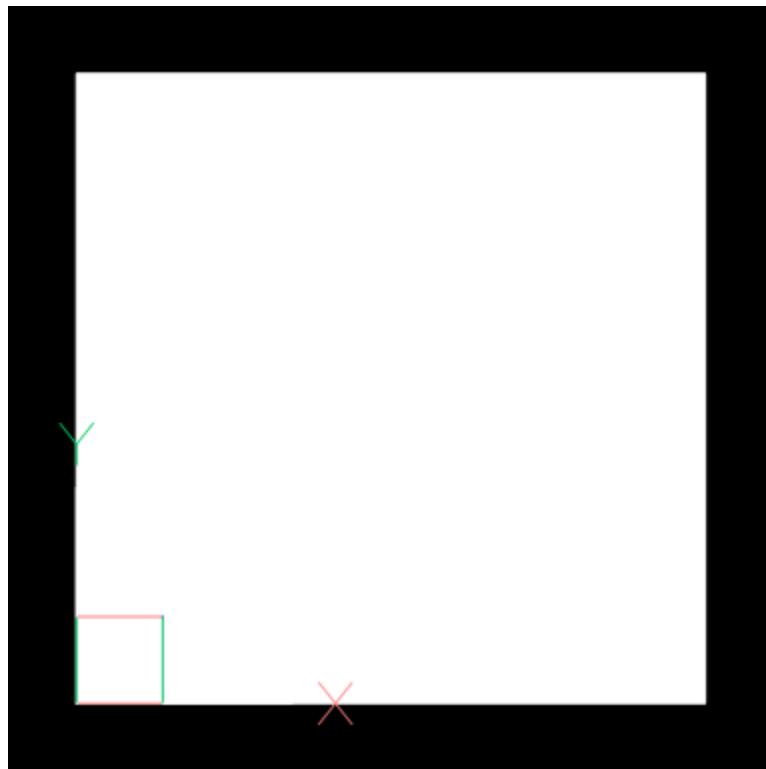
Trace

TRACE entity ([DXF Reference](#)) is solid filled triangle or quadrilateral. Access vertices by name (`entity.dxf.vtx0 = (1.7, 2.3)`) or by index (`entity[0] = (1.7, 2.3)`). If only 3 vertices are provided the last (3rd) vertex will be repeated in the DXF file.

The TRACE entity stores the vertices in an unusual way, the last two vertices are reversed. The coordinates `[(0, 0), (1, 0), (1, 1), (0, 1)]` do not create a square as you would expect:



Reverse the last two vertices to get the *expected* square: `[(0, 0), (1, 0), (0, 1), (1, 1)]`



Note: The `Trace.vertices()` and `Trace.wcs_vertices()` methods return the vertices in the *expected* (reversed) order.

Subclass of	<code>ezdxf.entities.DXFGraphic</code>
DXF type	'TRACE'
Factory function	<code>ezdxf.layouts.BaseLayout.add_trace()</code>
Inherited DXF attributes	<code>Common graphical DXF attributes</code>

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

```
class eздxf.entities.Trace

    dxf.vtx0
        Location of 1. vertex (2D/3D Point in OCS)
    dxf.vtx1
        Location of 2. vertex (2D/3D Point in OCS)
    dxf.vtx2
        Location of 3. vertex (2D/3D Point in OCS)
    dxf.vtx3
        Location of 4. vertex (2D/3D Point in OCS)
transform(m: Matrix44) → Trace
    Transform the SOLID/TRACE entity by transformation matrix m inplace.
```

vertices (*close: bool=False*) → List[Vec3]

Returns OCS vertices in correct order, if argument *close* is True, last vertex == first vertex. Does **not** return duplicated last vertex if represents a triangle.

New in version 0.15.

wcs_vertices (*close: bool=False*) → List[Vec3]

Returns WCS vertices in correct order, if argument *close* is True, last vertex == first vertex. Does **not** return duplicated last vertex if represents a triangle.

New in version 0.15.

Underlay

UNDERLAY entity ([DXF Reference](#)) links an underlay file to the DXF file, the file itself is not embedded into the DXF file, it is always a separated file. The (PDF)UNDERLAY entity is like a block reference, you can use it multiple times to add the underlay on different locations with different scales and rotations. But therefore you need a also a (PDF)DEFINITION entity, see [UnderlayDefinition](#).

The DXF standard supports three different file formats: PDF, DWF (DWFx) and DGN. An Underlay can be clipped by a rectangle or a polygon path. The clipping coordinates are 2D *OCS* coordinates in drawing units but without scaling.

Subclass of	<i>ezdxf.entities.DXFGraphic</i>
DXF type	internal base class
Factory function	<i>ezdxf.layouts.BaseLayout.add_underlay()</i>
Inherited DXF attributes	Common graphical DXF attributes
Required DXF version	DXF R2000 ('AC1015')

class *ezdxf.entities.Underlay*

Base class of *PdfUnderlay*, *DwfUnderlay* and *DgnUnderlay*

dxf.insert

Insertion point, lower left corner of the image in *OCS*.

dxf.scale_x

Scaling factor in x-direction (float)

dxf.scale_y

Scaling factor in y-direction (float)

dxf.scale_z

Scaling factor in z-direction (float)

dxf.rotation

ccw rotation in degrees around the extrusion vector (float)

dxf.extrusion

extrusion vector, default = (0, 0, 1)

dxf.underlay_def_handle

Handle to the underlay definition entity, see [UnderlayDefinition](#)

dxf.flags

dx.f.flags	Value	Description
UNDERLAY_CLIPPING	1	clipping is on/off
UNDERLAY_ON	2	underlay is on/off
UNDERLAY_MONOCHROME	4	Monochrome
UNDERLAY_ADJUST_FOR_BACKGROUND	8	Adjust for background

dx.f.contrast

Contrast value (20 - 100; default = 100)

dx.f.fade

Fade value (0 - 80; default = 0)

clipping

True or False (read/write)

on

True or False (read/write)

monochrome

True or False (read/write)

adjust_for_background

True or False (read/write)

scale

Scaling (x, y, z) tuple (read/write)

boundary_path

Boundary path as list of vertices (read/write).

Two vertices describe a rectangle (lower left and upper right corner), more than two vertices is a polygon as clipping path.

get_underlay_def() → UnderlayDefinitionReturns the associated DEFINITION entity. see [UnderlayDefinition](#).**set_underlay_def(underlay_def: UnderlayDefinition)** → NoneSet the associated DEFINITION entity. see [UnderlayDefinition](#).**reset_boundary_path()**

Removes the clipping path.

PdfUnderlay

Subclass of	ezdxf.entities.Underlay
DXF type	'PDFUNDERLAY'
Factory function	ezdxf.layouts.BaseLayout.add_underlay()
Inherited DXF attributes	Common graphical DXF attributes
Required DXF version	DXF R2000 ('AC1015')

class ezdxf.entities.PdfUnderlay

PDF underlay.

DwfUnderlay

Subclass of	<code>ezdxf.entities.Underlay</code>
DXF type	'DWFUNDERLAY'
Factory function	<code>ezdxf.layouts.BaseLayout.add_underlay()</code>
Inherited DXF attributes	<i>Common graphical DXF attributes</i>
Required DXF version	DXF R2000 ('AC1015')

```
class ezdxf.entities.DwfUnderlay
    DWF underlay.
```

DgnUnderlay

Subclass of	<code>ezdxf.entities.Underlay</code>
DXF type	'DGNUNDERLAY'
Factory function	<code>ezdxf.layouts.BaseLayout.add_underlay()</code>
Inherited DXF attributes	<i>Common graphical DXF attributes</i>
Required DXF version	DXF R2000 ('AC1015')

```
class ezdxf.entities.DgnUnderlay
    DGN underlay.
```

Viewport

The VIEWPORT (DXF Reference) entity is a window from a paperspace layout to the modelspace.

Subclass of	<code>ezdxf.entities.DXFGraphic</code>
DXF type	'VIEWPORT'
Factory function	<code>ezdxf.layouts.Paperspace.add_viewport()</code>
Inherited DXF attributes	<i>Common graphical DXF attributes</i>

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

```
class ezdxf.entities.Viewport
```

dx.f.center

Center point of the viewport located in the paper space layout in paper space units stored as 3D point.
(Error in the DXF reference)

dx.f.width

Viewport width in paperspace units (float)

dx.f.height

Viewport height in paperspace units (float)

dx.f.status

Viewport status field (int)

-1	On, but is fully off screen, or is one of the viewports that is not active because the \$MAXACTVP count is currently being exceeded.
0	Off
>0	On and active. The value indicates the order of stacking for the viewports, where 1 is the active viewport, 2 is the next, and so forth

```

dx.f.id
    Viewport id (int)

dx.f.view_center_point
    View center point in modelspace stored as 2D point, but represents a WCS point. (Error in the DXF reference)

dx.f.snap_base_point
dx.f.snap_spacing
dx.f.snap_angle
dx.f.grid_spacing
dx.f.view_direction_vector
    View direction (3D vector in WCS).

dx.f.view_target_point
    View target point (3D point in WCS).

dx.f.perspective_lens_length
    Lens focal length in mm as 35mm film equivalent.

dx.f.front_clip_plane_z_value
dx.f.back_clip_plane_z_value

dx.f.view_height
    View height in WCS.

dx.f.view_twist_angle
dx.f.circle_zoom

dx.f.flags
    Viewport status bit-coded flags:

```

Bit value	Constant in ezdxf.const	Description
1 (0x1)	VSF_PERSPECTIVE_MODE	perspective mode
2 (0x2)	VSF_FRONT_CLIPPING	Enables front clipping
4 (0x4)	VSF_BACK_CLIPPING	Enables back clipping
8 (0x8)	VSF_UCS_FOLLOW	Enables UCS follow
16 (0x10)	VSF_FRONT_CLIPPING_AT_EYE	Front clipping at eye
32 (0x20)	VSF_UCS_ICON_VISIBILITY	UCS icon visibility
64 (0x40)	VSF_UCS_ICON_AT_ORIGIN	UCS icon at origin
128 (0x80)	VSF_FAST_ZOOM	Enables fast zoom
256 (0x100)	VSF_SNAP_MODE	Enables snap mode
512 (0x200)	VSF_GRID_MODE	Enables grid mode
1024 (0x400)	VSF_ISOMETRIC_SNAP_STYLE	Isometric snap style
2048 (0x800)	VSF_HIDE_PLOT_MODE	Hides plot mode
4096 (0x1000)	VSF_KISOPAIR_TOP	kIsoPairTop. If set and kIsoPairRight is not set, then isopair top is enabled. If both kIsoPairTop and kIsoPairRight are set, then isopair left is enabled
8192 (0x2000)	VSF_KISOPAIR_RIGHT	kIsoPairRight. If set and kIsoPairTop is not set, then isopair right is enabled
16384 (0x4000)	VSF_LOCK_ZOOM	Enables viewport zoom locking
32768 (0x8000)	VSF_CURRENTLY_ALWAYS_ENABLED	Always enabled
65536 (0x10000)	VSF_NON_RECTANGULAR_CLIPPING	Nongular clipping
131072 (0x20000)	VSF_TURN_VIEWPORT_OFF	Viewport off
262144 (0x40000)	VSF_NO_GRID_LIMITS	Hides the display of the grid beyond the drawing limits
524288 (0x80000)	VSF_ADAPTIVE_GRID_DISPLAY	Adaptive grid display
1048576 (0x100000)	VSF_SUBDIVIDE_GRID	Hides subdivision of the grid below the set grid spacing when the grid display is adaptive
2097152 (0x200000)	VSF_GRID_FOLLOW_WORKPLANE	Follows workplane switching

Use helper method `set_flag_state()` to set and clear viewport flags, e.g. lock viewport:

```
vp.set_flag_state(ezdxr.const.VSF_LOCK_ZOOM, True)
```

```
dxr.clipping_boundary_handle
dxr.plot_style_name
dxr.render_mode
```

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

dx.dxf.ucs_per_viewport**dx.dxf.ucs_icon****dx.dxf.ucs_origin**

UCS origin as 3D point.

dx.dxf.ucs_x_axis

UCS x-axis as 3D vector.

dx.dxf.ucs_y_axis

UCS y-axis as 3D vector.

dx.dxf.ucs_handle

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

dx.dxf.ucs_ortho_type

0	not orthographic
1	Top
2	Bottom
3	Front
4	Back
5	Left
6	Right

dx.dxf.ucs_base_handleHandle of UCSTable of base UCS if UCS is orthographic (`Viewport.dx.dxf.ucs_ortho_type` is non-zero). If not present and `Viewport.dx.dxf.ucs_ortho_type` is non-zero, then base UCS is taken to be WORLD.**dx.dxf.elevation****dx.dxf.shade_plot_mode**

(DXF R2004)

0	As Displayed
1	Wireframe
2	Hidden
3	Rendered

dx.dxf.grid_frequency

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

dx.dxf.background_handle**dx.dxf.shade_plot_handle**

```
dxf.visual_style_handle  
dxf.default_lighting_flag  
dxf.default_lighting_style
```

0	One distant light
1	Two distant lights

```
dxf.view_brightness  
dxf.view_contrast  
dxf.ambient_light_color_1  
    as AutoCAD Color Index (ACI)  
dxf.ambient_light_color_2  
    as true color value  
dxf.ambient_light_color_3  
    as true color value  
dxf.sun_handle  
dxf.ref_vp_object_1  
dxf.ref_vp_object_2  
dxf.ref_vp_object_3  
dxf.ref_vp_object_4  
frozen_layers  
Set/get frozen layers as list of layer names.
```

Wipeout

THE WIPEOUT ([DXF Reference](#)) entity is a polygonal area that masks underlying objects with the current background color. The WIPEOUT entity is based on the IMAGE entity, but usage does not require any knowledge about the IMAGE entity.

The handles to the support entities *ImageDef* and *ImageDefReactor* are always “0”, both are not needed by the WIPEOUT entity.

Subclass of	<i>ezdxf.entities.Image</i>
DXF type	'WIPEOUT'
Factory function	<i>ezdxf.layouts.BaseLayout.add_wipeout()</i>
Inherited DXF attributes	<i>Common graphical DXF attributes</i>
Required DXF version	DXF R2000 ('AC1015')

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

```
class ezdxf.entities.Wipeout
```

set_masking_area (*vertices: Iterable[Vertex]*) → None
Set a new masking area, the area is placed in the layout xy-plane.

XLine

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

Subclass of	<code>ezdxf.entities.DXFGraphic</code>
DXF type	'XLINE'
Factory function	<code>ezdxf.layouts.BaseLayout.add_xline()</code>
Inherited DXF attributes	Common graphical DXF attributes
Required DXF version	DXF R2000 ('AC1015')

```
class ezdxf.entities.XLine

    dx.start
    Location point of line as (3D Point in WCS)
    dx.unit_vector
    Unit direction vector as (3D Point in WCS)
    transform(m: Matrix44) → XLine
        Transform the XLINE/RAY entity by transformation matrix m inplace.
    translate(dx: float, dy: float, dz: float) → XLine
        Optimized XLINE/RAY translation about dx in x-axis, dy in y-axis and dz in z-axis.
```

DXF Objects

All DXF objects can only reside in the OBJECTS section of a DXF document.

Dictionary

The **DICTIONARY** is a general storage entity.

AutoCAD maintains items such as MLINE_STYLES and GROUP definitions as objects in dictionaries. Other applications are free to create and use their own dictionaries as they see fit. The prefix '**ACAD_**' is reserved for use by AutoCAD applications.

Dictionary entries are (*key, DXFentity*) pairs for fully loaded or new created DXF documents. The referenced entities are owned by the dictionary and cannot be graphical entities that always belong to the layout in which they are located.

Loading DXF files is done in two passes, because at the first loading stage not all referenced objects are already stored in the entity database. Therefore the entities are stored as handle strings at the first loading stage and have to be replaced by the real entity at the second loading stage. If the entity is still a handle string after the second loading stage, the entity does not exist.

Dictionary keys are handled case insensitive by AutoCAD, but not by *ezdxf*, in doubt use an uppercase key. AutoCAD stores all keys in uppercase.

Subclass of	<code>ezdxf.entities.DXFObject</code>
DXF type	'DICTIONARY'
Factory function	<code>ezdxf.sections.objects.ObjectsSection.add_dictionary()</code>

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

class `ezdxf.entities.Dictionary`

`dxf.hard_owned`

If set to 1, indicates that elements of the dictionary are to be treated as hard-owned.

`dxf.cloning`

Duplicate record cloning flag (determines how to merge duplicate entries, ignored by *ezdxf*):

0	not applicable
1	keep existing
2	use clone
3	<xref>\$0\$<name>
4	\$0\$<name>
5	Unmangle name

`is_hard_owner`

Returns `True` if the dictionary is hard owner of entities. Hard owned entities will be destroyed by deleting the dictionary.

`__len__()` → int

Returns count of dictionary entries.

`__contains__(key: str) → bool`

Returns `key` in self.

`__getitem__(key: str) → DXFEntity`

Return `self[key]`.

The returned value can be a handle string if the entity does not exist.

Raises `DXFKeyError` – `key` does not exist

`__setitem__(key: str, value: DXFEntity) → None`

Set `self[key] = entity`.

Only DXF objects stored in the OBJECTS section are allowed as content of `Dictionary` objects. DXF entities stored in layouts are not allowed.

Raises `DXFTypeError` – invalid DXF type

`__delitem__(key: str) → None`

Delete `self[key]`.

Raises `DXFKeyError` – `key` does not exist

`keys()` → KeysView

Returns a `KeysView` of all dictionary keys.

`items()` → ItemsView

Returns an `ItemsView` for all dictionary entries as (key, entity) pairs. An entity can be a handle string if the entity does not exist.

count () → int

Returns count of dictionary entries.

get (key: str, default: DXFEntity = None) → Optional[DXFEntity]

Returns the *DXFEntity* for *key*, if *key* exist else *default*. An entity can be a handle string if the entity does not exist.

add (key: str, value: DXFEntity) → None

Add entry (*key*, *value*).

Raises

- DXFValueError – invalid entity handle
- DXFTypError – invalid DXF type

remove (key: str) → None

Delete entry *key*. Raises DXFKeyError, if *key* does not exist. Destroys hard owned DXF entities.

discard (key: str) → None

Delete entry *key* if exists. Does not raise an exception if *key* doesn't exist and does not destroy hard owned DXF entities.

clear () → None

Delete all entries from the dictionary and destroys hard owned DXF entities.

add_new_dict (key: str, hard_owned: bool = False) → Dictionary

Create a new sub-dictionary of type *Dictionary*.

Parameters

- **key** – name of the sub-dictionary
- **hard_owned** – entries of the new dictionary are hard owned

get_required_dict (key: str) → Dictionary

Get entry *key* or create a new *Dictionary*, if *Key* not exist.

add_dict_var (key: str, value: str) → DictionaryVar

Add a new *DictionaryVar*.

Parameters

- **key** – entry name as string
- **value** – entry value as string

add_xrecord (key: str) → XRecord

Add a new *XRecord*.

Parameters key – entry name as string**link_dxf_object (name: str, obj: DXFEntity) → None**

Add *obj* and set owner of *obj* to this dictionary.

Graphical DXF entities have to reside in a layout and therefore can not be owned by a *Dictionary*.

Raises DXFTypError – *obj* has invalid DXF type

DictionaryWithDefault

Subclass of	<code>ezdxf.entities.Dictionary</code>
DXF type	'ACDBDICTIONARYWDFLT'
Factory function	<code>ezdxf.sections.objects.ObjectsSection.add_dictionary_with_default()</code>

```
class ezdxf.entities.DictionaryWithDefault
```

`dxfl.default`

Handle to default entry as hex string like FF00.

`get(key: str) → DXFEntity`

Returns `DXFEntity` for `key` or the predefined dictionary wide `dxfl.default` entity if `key` does not exist or `None` if default value also not exist.

`set_default(default: ezdxf.entities.dxfo.DXFObject) → None`

Set dictionary wide default entry.

Parameters `default` – default entry as `DXFEntity`

DictionaryVar

Subclass of	<code>ezdxf.entities.DXFObject</code>
DXF type	'DICTIONARYVAR'
Factory function	<code>ezdxf.entities.Dictionary.add_dict_var()</code>

```
class ezdxf.entities.DictionaryVar
```

`dxfl.schema`

Object schema number (currently set to 0)

`dxfl.value`

Value as string.

DXFLayout

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

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

```
class ezdxf.entities.DXFLLayout
```

`dxfl.name`

Layout name as shown in tabs by `CAD` applications

TODO

DXFObject

Common base class for all non-graphical DXF objects.

class ezdxf.entities.**DXFObject**

A class hierarchy marker class and subclass of `ezdxf.entities.DXFEntity`

GeoData

The **GEODATA** entity is associated to the `Modelspace` object. The **GEODATA** entity is supported since the DXF version R2000, but was officially documented the first time in the DXF reference for version R2009.

Subclass of	<code>ezdxf.entities.DXFObject</code>
DXF type	'GEODATA'
Factory function	<code>ezdxf.layouts.Modelspace.new_geodata()</code>
Required DXF version	R2010 ('AC1024')

See also:

`using_geodata.py`

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

class ezdxf.entities.**GeoData**

`dxf.version`

1	R2009
2	R2010

`dxf.coordinate_type`

0	unknown
1	local grid
2	projected grid
3	geographic (latitude/longitude)

`dxf.block_record_handle`

Handle of host BLOCK_RECORD table entry, in general the `Modelspace`.

Changed in version 0.10: renamed from `dxf.block_record`

`dxf.design_point`

Reference point in `WCS` coordinates.

`dxf.reference_point`

Reference point in geo coordinates, valid only when coordinate type is *local grid*. The difference between `dxf.design_point` and `dxf.reference_point` defines the translation from WCS coordinates to geo-coordinates.

dx_f.north_direction

North direction as 2D vector. Defines the rotation (about the *dx_f.design_point*) to transform from WCS coordinates to geo-coordinates

dx_f.horizontal_unit_scale

Horizontal unit scale, factor which converts horizontal design coordinates to meters by multiplication.

dx_f.vertical_unit_scale

Vertical unit scale, factor which converts vertical design coordinates to meters by multiplication.

dx_f.horizontal_units

Horizontal units (see *BlockRecord*). Will be 0 (Unitless) if units specified by horizontal unit scale is not supported by AutoCAD enumeration.

dx_f.vertical_units

Vertical units (see *BlockRecord*). Will be 0 (Unitless) if units specified by vertical unit scale is not supported by AutoCAD enumeration.

dx_f.up_direction

Up direction as 3D vector.

dx_f.scale_estimation_method

1	none
2	user specified scale factor
3	grid scale at reference point
4	prismoidal

dx_f.sea_level_correction

Bool flag specifying whether to do sea level correction.

dx_f.user_scale_factor**dx_f.sea_level_elevation****dx_f.coordinate_projection_radius****dx_f.geo_rss_tag****dx_f.observation_from_tag****dx_f.observation_to_tag****dx_f.mesh_faces_count****source_vertices**

2D source vertices in the CRS of the GeoData as *VertexArray*. Used together with *target_vertices* to define the transformation from the CRS of the GeoData to WGS84.

target_vertices

2D target vertices in WGS84 (EPSG:4326) as *VertexArray*. Used together with *source_vertices* to define the transformation from the CRS of the geoData to WGS84.

faces

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

coordinate_system_definition

The coordinate system definition string. Stored as XML. Defines the CRS used by the GeoData. The EPSG number and other details like the axis-ordering of the CRS is stored.

get_crs() → Tuple[int, bool]

Returns the EPSG index and axis-ordering, axis-ordering is True if first axis is labeled “E” or “W” and False if first axis is labeled “N” or “S”.

If axis-ordering is False the CRS is not compatible with the `__geo_interface__` or GeoJSON (see chapter 3.1.1).

Raises InvalidGeoDataException – for invalid or unknown XML data

The EPSG number is stored in a tag like:

```
<Alias id="27700" type="CoordinateSystem">
    <ObjectId>OSGB1936.NationalGrid</ObjectId>
    <Namespace>EPSG Code</Namespace>
</Alias>
```

The axis-ordering is stored in a tag like:

```
<Axis uom="METER">
    <CoordinateSystemAxis>
        <AxisOrder>1</AxisOrder>
        <AxisName>Easting</AxisName>
        <AxisAbbreviation>E</AxisAbbreviation>
        <AxisDirection>east</AxisDirection>
    </CoordinateSystemAxis>
    <CoordinateSystemAxis>
        <AxisOrder>2</AxisOrder>
        <AxisName>Northing</AxisName>
        <AxisAbbreviation>N</AxisAbbreviation>
        <AxisDirection>north</AxisDirection>
    </CoordinateSystemAxis>
</Axis>
```

get_crs_transformation(no_checks: bool = False) → Tuple[Matrix44, int]

Returns the transformation matrix and the EPSG index to transform WCS coordinates into CRS coordinates. Because of the lack of proper documentation this method works only for tested configurations, set argument `no_checks` to True to use the method for untested geodata configurations, but the results may be incorrect.

Supports only “Local Grid” transformation!

Raises InvalidGeoDataException – for untested geodata configurations

setup_local_grid(design_point: Vec3, reference_point: Vec3, north_direction: Vec2=Y_AXIS, crs: str=EPSG_3395)

Setup local grid coordinate system. This method is designed to setup CRS similar to *EPSG:3395 World Mercator*, the basic features of the CRS should fulfill this assumptions:

- base unit of reference coordinates is 1 meter
- right-handed coordinate system: +Y=north/+X=east/+Z=up

The CRS string is not validated nor interpreted!

Hint: The reference point must be a 2D cartesian map coordinate and not a globe (lon/lat) coordinate like stored in GeoJSON or GPS data.

Parameters

- `design_point` – WCS coordinates of the CRS reference point

- **reference_point** – CRS reference point in 2D cartesian coordinates
- **north_direction** – north direction a 2D vertex, default is (0, 1)
- **crs** – Coordinate Reference System definition XML string, default is the definition string for *EPSG:3395 World Mercator*

ImageDef

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

Subclass of	<code>ezdxf.entities.DXFObject</code>
DXF type	'IMAGEDEF'
Factory function (1)	<code>ezdxf.document.Drawing.add_image_def()</code>
Factory function (2)	<code>ezdxf.sections.objects.ObjectsSection.add_image_def()</code>

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

class `ezdxf.entities.ImageDef`

`dx.dxf.class_version`

Current version is 0.

`dx.dxf.filename`

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

`dx.dxf.image_size`

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

`dx.dxf.pixel_size`

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

`dx.dxf.loaded`

0 = unloaded; 1 = loaded, default = 1

`dx.dxf.resolution_units`

0	No units
2	Centimeters
5	Inch

Default = 0

ImageDefReactor

class `ezdxf.entities.ImageDefReactor`

`dx.dxf.class_version`

`dx.dxf.image_handle`

MLeaderStyle

The MLEADERSTYLE object ([DXF Reference](#)) store all attributes required to create new *MultiLeader* entities. The meaning of these attributes are not really documented in the [DXF Reference](#). The default style “Standard” always exist.

See also:

- `ezdxf.entities.MultiLeader`
- `ezdxf.render.MultiLeaderBuilder`
- *Tutorial for MultiLeader*

Create a new *MLeaderStyle*:

```
import ezdxf

doc = ezdxf.new()
new_style = doc.mleader_styles.new("NewStyle")
```

Duplicate an existing style:

```
duplicated_style = doc.mleader_styles.duplicate_entry("Standard", "DuplicatedStyle")
```

Subclass of	<code>ezdxf.entities.DXFObject</code>
DXF type	'MLEADERSTYLE'
Factory function	<code>ezdxf.document.Drawing.mleader_styles.new()</code>

`class ezdxf.entities.MLeaderStyle`

<code>dx.f.align_space</code>	unknown meaning
<code>dx.f.arrow_head_handle</code>	handle of default arrow head, see also <code>ezdxf.render.arrows</code> module, by default no handle is set, which mean default arrow “closed filled”
<code>dx.f.arrow_head_size</code>	default arrow head size in drawing units, default is 4.0
<code>dx.f.block_color</code>	default block color as ;term:raw color value, default is BY_BLOCK_RAW_VALUE
<code>dx.f.block_connection_type</code>	

0	center extents
1	insertion point

<code>dx.f.block_record_handle</code>	handle to block record of the BLOCK content, not set by default
<code>dx.f.block_rotation</code>	default BLOCK rotation in radians, default is 0.0
<code>dx.f.block_scale_x</code>	default block x-axis scale factor, default is 1.0

`dx.dxf.block_scale_y`
default block y-axis scale factor, default is 1.0

`dx.dxf.block_scale_z`
default block z-axis scale factor, default is 1.0

`dx.dxf.break_gap_size`
default break gap size, default is 3.75

`dx.dxf.char_height`
default MTEXT char height, default is 4.0

`dx.dxf.content_type`

0	none
1	BLOCK
2	MTEXT
3	TOLERANCE

default is MTEXT (2)

`dx.dxf.default_text_content`
default MTEXT content as string, default is ""

`dx.dxf.dogleg_length`
default dogleg length, default is 8.0

`dx.dxf.draw_leader_order_type`
unknown meaning

`dx.dxf.draw_mleader_order_type`
unknown meaning

`dx.dxf.first_segment_angle_constraint`
angle of fist leader segment in radians, default is 0.0

`dx.dxf.has_block_rotation`

`dx.dxf.has_block_scaling`

`dx.dxf.has_dogleg`
default is 1

`dx.dxf.has_landing`
default is 1

`dx.dxf.is_annotation`
default is 0

`dx.dxf.landing_gap`
default landing gap size, default is 2.0

`dx.dxf.leader_line_color`
default leader line color as *raw color* value, default is BY_BLOCK_RAW_VALUE

`dx.dxf.leader_linetype_handle`
handle of default leader linetype

`dx.dxf.leader_lineweight`
default leader linewidth, default is LINEWEIGHT_BYBLOCK

dx_f.leader_type

0	invisible
1	straight line leader
2	spline leader

default is 1

dx_f.max_leader_segments_points

max count of leader segments, default is 2

dx_f.name

MLEADERSTYLE name

dx_f.overwrite_property_value

unknown meaning

dx_f.scale

overall scaling factor, default is 1.0

dx_f.second_segment_angle_constraint

angle of fist leader segment in radians, default is 0.0

dx_f.text_align_always_left

use always left side to attach leaders, default is 0

dx_f.text_alignment_type

unknown meaning - its not the MTEXT attachment point!

dx_f.text_angle_type

0	text angle is equal to last leader line segment angle
---	---

1	text is horizontal
---	--------------------

2	text angle is equal to last leader line segment angle, but potentially rotated by 180 degrees so the right side is up for readability.
---	--

default is 1

dx_f.text_attachment_direction

defines whether the leaders attach to the left & right of the content BLOCK/MTEXT or attach to the top & bottom:

0	horizontal - left & right of content
1	vertical - top & bottom of content

default is 0

dx_f.text_bottom_attachment_type

9	center
10	overline and center

default is 9

dx.dxf.text_color

default MTEXT color as *raw color* value, default is BY_BLOCK_RAW_VALUE

dx.dxf.text_left_attachment_type

0	top of top MTEXT line
1	middle of top MTEXT line
2	middle of whole MTEXT
3	middle of bottom MTEXT line
4	bottom of bottom MTEXT line
5	bottom of bottom MTEXT line & underline bottom MTEXT line
6	bottom of top MTEXT line & underline top MTEXT line
7	bottom of top MTEXT line
8	bottom of top MTEXT line & underline all MTEXT lines

dx.dxf.text_right_attachment_type

0	top of top MTEXT line
1	middle of top MTEXT line
2	middle of whole MTEXT
3	middle of bottom MTEXT line
4	bottom of bottom MTEXT line
5	bottom of bottom MTEXT line & underline bottom MTEXT line
6	bottom of top MTEXT line & underline top MTEXT line
7	bottom of top MTEXT line
8	bottom of top MTEXT line & underline all MTEXT lines

dx.dxf.text_style_handle

handle of the default MTEXT text style, not set by default, which means “Standard”

dx.dxf.text_top_attachment_type

9	center
10	overline and center

Placeholder

The ACDBPLACEHOLDER object for internal usage.

Subclass of	<code>ezdxf.entities.DXFObject</code>
DXF type	'ACDBPLACEHOLDER'
Factory function	<code>ezdxf.sections.objects.ObjectsSection.add_placeholder()</code>

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

```
class ezdxf.entities.Placeholder
```

PlotSettings

All PLOTSETTINGS attributes are part of the `DXFLayout` entity, I don't know if this entity also appears as standalone entity.

Subclass of	<code>ezdxf.entities.DXFOBJECT</code>
DXF type	'PLOTSETTINGS'
Factory function	internal data structure

```
class ezdxf.entities.PlotSettings
```

`dxfl.page_setup_name`
Page setup name

TODO

Sun

SUN entity defines properties of the sun.

Subclass of	<code>ezdxf.entities.DXFOBJECT</code>
DXF type	'SUN'
Factory function	creating a new SUN entity is not supported

```
class ezdxf.entities.Sun
```

`dxfl.version`
Current version is 1.

`dxfl.status`
on = 1 or off = 0

`dxfl.color`
AutoCAD Color Index (ACI) value of the sun.

`dxfl.true_color`
true color value of the sun.

`dxfl.intensity`
Intensity value in the range of 0 to 1. (float)

`dxfl.julian_day`
use `calendardate()` to convert `dxfl.julian_day` to `datetime.datetime` object.

`dxfl.time`
Day time in seconds past midnight. (int)

`dxfl.daylight_savings_time`

`dxfl.shadows`

0	Sun do not cast shadows
1	Sun do cast shadows

```
dxf.shadow_type  
dxf.shadow_map_size  
dxf.shadow_softness
```

UnderlayDefinition

UnderlayDefinition (DXF Reference) defines an underlay file, which can be placed by the *Underlay* entity.

Subclass of	<code>ezdxf.entities.DXFObject</code>
DXF type	internal base class
Factory function (1)	<code>ezdxf.document.Drawing.add_underlay_def()</code>
Factory function (2)	<code>ezdxf.sections.objects.ObjectsSection.add_underlay_def()</code>

```
class ezdxf.entities.UnderlayDefinition  
Base class of PdfDefinition, DwfDefinition and DgnDefinition
```

`dxf.filename`

Relative (to the DXF file) or absolute path to the underlay file as string.

`dxf.name`

Defines which part of the underlay file to display.

'pdf'	PDF page number
'dgn'	always 'default'
'dwf'	?

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

PdfDefinition

Subclass of	<code>ezdxf.entities.UnderlayDefinition</code>
DXF type	'PDFDEFINITION'
Factory function (1)	<code>ezdxf.document.Drawing.add_underlay_def()</code>
Factory function (2)	<code>ezdxf.sections.objects.ObjectsSection.add_underlay_def()</code>

```
class ezdxf.entities.PdfDefinition  
PDF underlay file.
```

DwfDefinition

Subclass of	<code>ezdxf.entities.UnderlayDefinition</code>
DXF type	'DWFDEFINITION'
Factory function (1)	<code>ezdxf.document.Drawing.add_underlay_def()</code>
Factory function (2)	<code>ezdxf.sections.objects.ObjectsSection.add_underlay_def()</code>

```
class ezdxf.entities.DwfDefinition
    DWF underlay file.
```

DgnDefinition

Subclass of	<code>ezdxf.entities.UnderlayDefinition</code>
DXF type	'DGNDEFINITION'
Factory function (1)	<code>ezdxf.document.Drawing.add_underlay_def()</code>
Factory function (2)	<code>ezdxf.sections.objects.ObjectsSection.add_underlay_def()</code>

```
class ezdxf.entities.DgnDefinition
    DGN underlay file.
```

XRecord

Important class for storing application defined data in DXF files.

XRECORD objects are used to store and manage arbitrary data. They are composed of DXF group codes ranging from 1 through 369. This object is similar in concept to XDATA but is not limited by size or order.

To reference a XRECORD by an DXF entity, store the handle of the XRECORD in the XDATA section, application defined data or the ExtensionDict of the DXF entity.

See also:

- *Extended Data (XDATA)*
- *Extension Dictionary*
- *Storing Custom Data in DXF Files*

Subclass of	<code>ezdxf.entities.DXFObject</code>
DXF type	'XRECORD'
Factory function	<code>ezdxf.sections.objects.ObjectsSection.add_xrecord()</code>

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

```
class ezdxf.entities.XRecord
```

dxfs.cloning

Duplicate record cloning flag (determines how to merge duplicate entries, ignored by `ezdxf`):

0	not applicable
1	keep existing
2	use clone
3	<xref>\$0\$<name>
4	\$0\$<name>
5	Unmangle name

tags

Raw DXF tag container [Tags](#). Be careful *ezdxf* does not validate the content of XRECORDS.

clear() → None

Remove all DXF tags.

reset(tags: Iterable[Union[ezdxf.lldxftypes.DXFTag, Tuple[int, Any]]]) → None

Reset DXF tags.

extend(tags: Iterable[Union[ezdxf.lldxftypes.DXFTag, Tuple[int, Any]]]) → None

Extend DXF tags.

Extended Data (XDATA)

Extended data (XDATA) is a DXF tags structure to store arbitrary data in DXF entities. The XDATA is associated to an [AppID](#) and only one tag list is supported for each AppID per entity.

Warning: Low level usage of XDATA is an advanced feature, it is the responsibility of the programmer to create valid XDATA structures. Any errors can invalidate the DXF file!

This section shows how to store DXF tags directly in DXF entity but there is also a more user friendly and safer way to store custom XDATA in DXF entities:

- [XDataUserList](#)
- [XDataUserDict](#)

Use the high level methods of [DXFEntity](#) to manage XDATA tags.

- [has_xdata\(\)](#)
- [get_xdata\(\)](#)
- [set_xdata\(\)](#)

Get XDATA tags as a `ezdxf.lldxftags.Tags` data structure, **without** the mandatory first tag (1001, AppID):

```
if entity.has_xdata("EZDXF"):  
    tags = entity.get_xdata("EZDXF")  
  
# or use alternatively:  
try:  
    tags = entity.get_xdata("EZDXF")  
except DXFValueError:  
    # XDATA for "EZDXF" does not exist  
    ...
```

Set DXF tags as list of (group code, value) tuples or as `ezdxf.lldxftags.Tags` data structure, valid DXF tags for XDATA are documented in the section about the [Extended Data](#) internals. The mandatory first tag (1001, AppID) is inserted automatically if not present.

Set only new XDATA tags:

```
if not entity.has_xdata("EZDXF"):  
    entity.set_xdata("EZDXF", [(1000, "MyString")])
```

Replace or set new XDATA tags:

```
entity.set_xdata("EZDXF", [(1000, "MyString")])
```

See also:

- Tutorial: [Storing Custom Data in DXF Files](#)
- Internals about [Extended Data](#) tags
- Internal XDATA management class: [XData](#)
- DXF R2018 Reference

Application-Defined Data (AppData)

The application-defined data feature is not very well documented in the DXF reference, so usage as custom data store is not recommended. AutoCAD uses these feature to store the handle to the extension dictionary ([ExtensionDict](#)) of a DXF entity and the handles to the persistent reactors ([Reactors](#)) of a DXF entity.

Use the high level methods of [DXFEntity](#) to manage application-defined data tags.

- [has_app_data\(\)](#)
- [get_app_data\(\)](#)
- [set_app_data\(\)](#)
- [discard_app_data\(\)](#)

Hint: Ezdxf uses special classes to manage the extension dictionary and the reactor handles. These features cannot be accessed by the methods above.

Set application-defined data:

```
entity.set_app_data("YOURAPPID", [(1, "DataString"))])
```

Setting the content tags can contain the opening structure tag (102, “{YOURAPPID”) and the closing tag (102, “}”), but doesn’t have to. The returned [Tags](#) objects does not contain these structure tags. Which tags are valid for application-defined data is not documented.

The AppID has to have an entry in the AppID table.

Get application-defined data:

```
if entity.has_app_data("YOURAPPID"):
    tags = entity.get_app_data("YOURAPPID")

# tags content is [DXFTag(1, 'DataString')]
```

See also:

- Internals about [Application-Defined Codes](#)
- Internal AppData management class: [AppData](#)

Extension Dictionary

Every entity can have an extension dictionary, which can reference arbitrary DXF objects from the OBJECTS section but not graphical entities. Using this mechanism, several applications can attach data to the same entity. The usage of

extension dictionaries is more complex than *Extended Data (XDATA)* but also more flexible with higher capacity for adding data.

Use the high level methods of `DXFEntity` to manage extension dictionaries.

- `has_extension_dict()`
- `get_extension_dict()`
- `new_extension_dict()`
- `discard_extension_dict()`

The main data storage objects referenced by extension dictionaries are:

- `Dictionary`, structural container
- `DictionaryVar`, stores a single string
- `XRecord`, stores arbitrary data

See also:

- Tutorial: [Storing Custom Data in DXF Files](#)

class `ezdxf.entities.xdict.ExtensionDict`
Internal management class for extension dictionaries.

See also:

- Underlying DXF `Dictionary` class
- DXF Internals: [Extension Dictionary](#)
- [DXF R2018 Reference](#)

`__contains__(key: str)`
Return `key` in self.

`__getitem__(key: str)`
Get `self[key]`.

`__setitem__(key: str, value)`
Set `self[key]` to value.

Only DXF objects stored in the OBJECTS section are allowed as content of the extension dictionary. DXF entities stored in layouts are not allowed.

Raises `DXFTypeError` – invalid DXF type

`__delitem__(key: str)`
Delete `self[key]`, destroys referenced entity.

`__len__()`
Returns count of extension dictionary entries.

`get(key: str, default=None) → Optional[DXFEntity]`
Return extension dictionary entry `key`.

`keys()`
Returns a `KeysView` of all extension dictionary keys.

`items()`
Returns an `ItemsView` for all extension dictionary entries as (key, entity) pairs. An entity can be a handle string if the entity does not exist.

discard (*key: str*) → None
Discard extension dictionary entry *key*.

add_dictionary (*name: str, hard_owned: bool = True*) → Dictionary
Create a new *Dictionary* object as extension dictionary entry *name*.

add_dictionary_var (*name: str, value: str*) → DictionaryVar
Create a new *DictionaryVar* object as extension dictionary entry *name*.

add_xrecord (*name: str*) → XRecord
Create a new *XRecord* object as extension dictionary entry *name*.

link_dxf_object (*name: str, obj: DXFObject*) → None
Link *obj* to the extension dictionary as entry *name*.
Linked objects are owned by the extensions dictionary and therefore cannot be a graphical entity, which have to be owned by a *BaseLayout*.

Raises DXFTypeError – *obj* has invalid DXF type

destroy()
Destroy the underlying *Dictionary* object.

Reactors

Persistent reactors are optional object handles of objects registering themselves as reactors on an object. Any DXF object or DXF entity may have reactors.

Use the high level methods of *DXFEntity* to manage persistent reactor handles.

- *has_reactors()*
- *get_reactors()*
- *set_reactors()*
- *append_reactor_handle()*
- *discard_reactor_handle()*

*Ezdx*f keeps these reactors only up to date, if this is absolute necessary according to the DXF reference.

See also:

- Internals about *Persistent Reactors*
- Internal Reactors management class: *Reactors*

Block Reference Management

The package *ezdxf* is not designed as a CAD library and does not automatically monitor all internal changes. This enables faster entity processing at the cost of an unknown state of the DXF document.

In order to carry out precise BLOCK reference management, i.e. to handle dependencies or to delete unused BLOCK definition, the block reference status (counter) must be acquired explicitly by the package user. All block reference management structures must be explicitly recreated each time the document content is changed. This is not very efficient, but it is safe.

Warning: And even with all this careful approach, it is always possible to destroy a DXF document by deleting an absolutely necessary block definition.

Always remember that *ezdxf* is not intended or suitable as a basis for a CAD application!

New in version 0.18.

```
class ezdxf.blkrefs.BlockDefinitionIndex(doc: Drawing)
    Index of all BlockRecord entities representing real BLOCK definitions, excluding all BlockRecord entities defining model space or paper space layouts. External references (XREF) and XREF overlays are included.

    rebuild()
        Rebuild index from scratch.

    has_handle(handle: str) → bool
        Returns True if a BlockRecord for the given block record handle exist.

    by_handle(handle: str) → Optional[BlockRecord]
        Returns the BlockRecord for the given block record handle or None.

    has_name(name: str) → bool
        Returns True if a BlockRecord for the given block name exist.

    by_name(name: str) → Optional[BlockRecord]
        Returns BlockRecord for the given block name or None.
```

```
class ezdxf.blkrefs.BlockReferenceCounter(doc: Drawing, index: ezdxf.blkrefs.BlockDefinitionIndex = None)
```

Counts all block references in a DXF document.

Check if a block is referenced by any entity or any resource (DIMSYTLE, MLEADERSTYLE) in a DXF document:

```
import ezdxf
from ezdxf.blkrefs import BlockReferenceCounter

doc = ezdxf.readfile("your.dxf")
counter = BlockReferenceCounter(doc)
count = counter.by_name("XYZ")
print(f"Block 'XYZ' is referenced {count} times.")
```

by_handle(handle: str) → int
Returns the block reference count for a given *BlockRecord* handle.

by_name(block_name: str) → int
Returns the block reference count for a given block name.

Const

The module *ezdxf.llodxf.const*, is also accessible from the *ezdxf* namespace:

```
from ezdxf.llodxf.const import DXF12
import ezdxf

print(DXF12)
print(ezdxf.const.DXF12)
```

DXF Version Strings

Name	Version	Alias
DXF9	“AC1004”	“R9”
DXF10	“AC1006”	“R10”
DXF12	“AC1009”	“R12”
DXF13	“AC1012”	“R13”
DXF14	“AC1014”	“R14”
DXF2000	“AC1015”	“R2000”
DXF2004	“AC1018”	“R2004”
DXF2007	“AC1021”	“R2007”
DXF2010	“AC1024”	“R2010”
DXF2013	“AC1027”	“R2013”
DXF2018	“AC1032”	“R2018”

Exceptions

```

class ezdxf.lldxf.const.DXFError
    Base exception for all ezdxf exceptions.

class ezdxf.lldxf.const.DXFStructureError(DXFError)
class ezdxf.lldxf.const.DXFVersionError(DXFError)
    Errors related to features not supported by the chosen DXF Version

class ezdxf.lldxf.const.DXFValueError(DXFError)
class ezdxf.lldxf.const.DXFInvalidLineType(DXFValueError)
class ezdxf.lldxf.const.DXFBlockInUseError(DXFValueError)
class ezdxf.lldxf.const.DXFKeyError(DXFError)
class ezdxf.lldxf.const.DXFUndefinedBlockError(DXFKeyError)
class ezdxf.lldxf.const.DXFAttributeError(DXFError)
class ezdxf.lldxf.const.DXFIndexError(DXFError)
class ezdxf.lldxf.const.DXFTypeError(DXFError)
class ezdxf.lldxf.const.DXFTableEntryError(DXFValueError)

```

6.8.3 DXF Entity Creation

Layout Factory Methods

Recommended way to create DXF entities.

For all supported entities exist at least one factory method in the `ezdxf.layouts.BaseLayout` class. All factory methods have the prefix: `add_...`

```

import ezdxf

doc = ezdxf.new()

```

(continues on next page)

(continued from previous page)

```
msp = doc.modelspace()
msp.add_line((0, 0, 0), (3, 0, 0), dxftattribs={"color": 2})
```

Thematic Index of Layout Factory Methods

DXF Primitives

- `add_3dface()`
- `add_arc()`
- `add_circle()`
- `add_ellipse()`
- `add_hatch()`
- `add_image()`
- `add_leader()`
- `add_line()`
- `add_lwpolyline()`
- `add_mesh()`
- `add_mline()`
- `add_mpolygon()`
- `add_multileader_mtext()`
- `add_multileader_block()`
- `add_point()`
- `add_polyface()`
- `add_polyline2d()`
- `add_polyline3d()`
- `add_polymesh()`
- `add_ray()`
- `add_shape()`
- `add_solid()`
- `add_trace()`
- `add_wipeout()`
- `add_xline()`

Text Entities

- `add_attdef()`
- `add_mtext_dynamic_auto_height_columns()`

- `add_mtext_dynamic_manual_height_columns()`
- `add_mtext_static_columns()`
- `add_mtext()`
- `add_text()`

Spline Entity

- `add_cad_spline_control_frame()`
- `add_open_spline()`
- `add_rational_spline()`
- `add_spline_control_frame()`
- `add_spline()`

Block References and Underlays

- `add_arrow_blockref()`
- `add_auto_blockref()`
- `add_blockref()`
- `add_underlay()`

Viewport Entity

Only available in paper space layouts.

- `add_viewport()`

Dimension Entities

Linear Dimension

- `add_aligned_dim()`
- `add_linear_dim()`
- `add_multi_point_linear_dim()`

Radius and Diameter Dimension

- `add_diameter_dim_2p()`
- `add_diameter_dim()`
- `add_radius_dim_2p()`
- `add_radius_dim_cra()`
- `add_radius_dim()`

Angular Dimension

- `add_angular_dim_21()`

- `add_angular_dim_3p()`
- `add_angular_dim_arc()`
- `add_angular_dim_cra()`

Arc Dimension

- `add_arc_dim_3p()`
- `add_arc_dim_arc()`
- `add_arc_dim_cra()`

Ordinate Dimension

- `add_ordinate_dim()`
- `add_ordinate_x_dim()`
- `add_ordinate_y_dim()`

Miscellaneous

- `add_entity()`
- `add_foreign_entity()`
- `add_arrow()`

ACIS Entities

The creation of the required **ACIS** data has to be done by an external library!

- `add_3dsolid()`
- `add_body()`
- `add_extruded_surface()`
- `add_lofted_surface()`
- `add_region()`
- `add_revolved_surface()`
- `add_surface()`
- `add_swept_surface()`

See also:

Layout base class: `BaseLayout`

Factory Functions

Alternative way to create DXF entities for advanced `ezdxf` users.

The `ezdxf.entities.factory` module provides the `new()` function to create new DXF entities by their DXF name and a dictionary of DXF attributes. This will bypass the validity checks in the factory methods of the `BaseLayout` class.

This new created entities are virtual entities which are not assigned to any DXF document nor to any layout. Add the entity to a layout (and document) by the layout method `add_entity()`.

```
import ezdxf
from ezdxf.entities import factory

doc = ezdxf.new()
msp = doc.modelspace()
line = factory.new(
    "LINE",
    dxftattribs={
        "start": (0, 0, 0),
        "end": (3, 0, 0),
        "color": 2,
    },
)
msp.add_entity(line)
```

Direct Object Instantiation

For advanced developers with knowledge about the internal design of `ezdxf`.

Import the entity classes from sub-package `ezdxf.entities` and instantiate them. This will bypass the validity checks in the factory methods of the `BaseLayout` class and maybe additional required setup procedures for some entities - **study the source code!**.

Warning: A refactoring of the internal `ezdxf` structures will break your code.

This new created entities are virtual entities which are not assigned to any DXF document nor to any layout. Add the entity to a layout (and document) by the layout method `add_entity()`.

```
import ezdxf
from ezdxf.entities import Line

doc = ezdxf.new()
msp = doc.modelspace()
line = Line.new(
    dxftattribs={
        "start": (0, 0, 0),
        "end": (3, 0, 0),
        "color": 2,
    }
)
msp.add_entity(line)
```

6.8.4 Enums

TextEntityAlignment

`class ezdxf.enums.TextEntityAlignment`

Text alignment enum for the `Text`, `Attrib` and `AttDef` entities.

`LEFT`

```
CENTER
RIGHT
ALIGNED
MIDDLE
FIT
BOTTOM_LEFT
BOTTOM_CENTER
BOTTOM_RIGHT
MIDDLE_LEFT
MIDDLE_CENTER
MIDDLE_RIGHT
TOP_LEFT
TOP_CENTER
TOP_RIGHT
```

MTextEntityAlignment

```
class ezdxf.enums.MTextEntityAlignment
Text alignment enum for the MText entity.

TOP_LEFT
TOP_CENTER
TOP_RIGHT
MIDDLE_LEFT
MIDDLE_CENTER
MIDDLE_RIGHT
BOTTOM_LEFT
BOTTOM_CENTER
BOTTOM_RIGHT
```

MTextParagraphAlignment

```
class ezdxf.enums.MTextParagraphAlignment
An enumeration.

DEFAULT
LEFT
RIGHT
CENTER
JUSTIFIED
DISTRIBUTED
```

MTextFlowDirection

```
class ezdxf.enums.MTextFlowDirection
An enumeration.
```

```
LEFT_TO_RIGHT
TOP_TO_BOTTOM
BY_STYLE
```

MTextLineAlignment

```
class ezdxf.enums.MTextLineAlignment
An enumeration.
```

```
BOTTOM
MIDDLE
TOP
```

MTextStroke

```
class ezdxf.enums.MTextStroke
Combination of flags is supported: UNDERLINE + STRIKE_TROUGH
```

```
UNDERLINE
STRIKE_THROUGH
OVERLINE
```

MTextLineSpacing

```
class ezdxf.enums.MTextLineSpacing
An enumeration.
```

```
AT_LEAST
EXACT
```

MTextBackgroundColor

```
class ezdxf.enums.MTextBackgroundColor
An enumeration.
```

```
OFF
COLOR
WINDOW
CANVAS
```

InsertUnits

```
class ezdxf.enums.InsertUnits
    An enumeration.

    Unitless
    Inches
    Feet
    Miles
    Millimeters
    Centimeters
    Meters
    Kilometers
    Microinches
    Mils
    Yards
    Angstroms
    Nanometers
    Microns
    Decimeters
    Decameters
    Hectometers
    Gigameters
    AstronomicalUnits
    Lightyears
    Parsecs
    USSurveyFeet
    USSurveyInch
    USSurveyYard
    USSurveyMile
```

Measurement

```
class ezdxf.enums.Measurement
    An enumeration.

    Imperial
    Metric
```

LengthUnits

```
class ezdxf.enums.LengthUnits
    An enumeration.

    Scientific
    Decimal
    Engineering
    Architectural
    Fractional
```

AngularUnits

```
class ezdxf.enums.AngularUnits
    An enumeration.

    DecimalDegrees
    DegreesMinutesSeconds
    Grad
    Radians
```

SortEntities

```
class ezdxf.enums.SortEntities
    An enumeration.

    DISABLE
    SELECTION
        Sorts for object selection
    SNAP
        Sorts for object snap
    REDRAW
        Sorts for redraws; obsolete
    MSLIDE
        Sorts for MSLIDE command slide creation; obsolete
    REGEN
        Sorts for REGEN commands
    PLOT
        Sorts for plotting
    POSTSCRIPT
        Sorts for PostScript output; obsolete
```

ACI

```
class ezdxf.enums.ACI
    AutoCAD Color Index
```

```
BYBLOCK
BYLAYER
BYOBJECT
RED
YELLOW
GREEN
CYAN
BLUE
MAGENTA
BLACK
WHITE
```

6.8.5 Colors

Colors Module

This module provides functions and constants to manage all kinds of colors in DXF documents.

Converter Functions

```
ezdxf.colors.rgb2int(rgb: Tuple[int, int, int]) → int
    Combined integer value from (r, g, b) tuple.

ezdxf.colors.int2rgb(value: int) → Tuple[int, int, int]
    Split RGB integer value into (r, g, b) tuple.

ezdxf.colors.aci2rgb(index: int) → Tuple[int, int, int]
    Convert AutoCAD Color Index (ACI) into (r, g, b) tuple, based on default AutoCAD colors.

ezdxf.colors.luminance(color: Tuple[int, int, int]) → float
    Returns perceived luminance for a RGB color in the range [0.0, 1.0] from dark to light.

ezdxf.colors.decode_raw_color(value: int) → Tuple[int, Union[int, Tuple[int, int, int]]]
    Decode raw color value as tuple(type, Union[aci, (r, g, b)]), the true color value is a (r, g, b) tuple.

ezdxf.colors.decode_raw_color_int(value: int) → Tuple[int, int]
    Decode raw color value as tuple(type, int), the true color value is a 24-bit int value.

ezdxf.colors.encode_raw_color(value: Union[int, Tuple[int, int, int]]) → int
    Encode true color value or AutoCAD Color Index (ACI) color value into a :term: raw color value.

ezdxf.colors.transparency2float(value: int) → float
    Returns transparency value as float from 0 to 1, 0 for no transparency (opaque) and 1 for 100% transparency.

    Parameters value – DXF integer transparency value, 0 for 100% transparency and 255 for opaque

ezdxf.colors.float2transparency(value: float) → int
    Returns DXF transparency value as integer in the range from 0 to 255, where 0 is 100% transparent and 255 is opaque.

    Parameters value – transparency value as float in the range from 0 to 1, where 0 is opaque and 1 is 100% transparent.
```

ACI Color Values

Common *AutoCAD Color Index (ACI)* values, also accessible as IntEnum `ezdxf.enums.ACIL`

BYBLOCK	0
BYLAYER	256
BYOBJECT	257
RED	1
YELLOW	2
GREEN	3
CYAN	4
BLUE	5
MAGENTA	6
BLACK (on light background)	7
WHITE (on dark background)	7

Default Palettes

Default color mappings from *AutoCAD Color Index (ACI)* to *true color* values.

model space	DXF_DEFAULT_COLORS
paper space	DXF_DEFAULT_PAPERSPACE_COLORS

Raw Color Types

COLOR_TYPE_BY_LAYER	0xC0
COLOR_TYPE_BY_BLOCK	0xC1
COLOR_TYPE_RGB	0xC2
COLOR_TYPE_ACI	0xC3
COLOR_TYPE_WINDOW_BG	0xC8

Raw Color Vales

BY_LAYER_RAW_VALUE	-1073741824
BY_BLOCK_RAW_VALUE	-1056964608
WINDOW_BG_RAW_VALUE	-939524096

Transparency Values

OPAQUE	0x20000FF
TRANSPARENCY_10	0x20000E5
TRANSPARENCY_20	0x20000CC
TRANSPARENCY_30	0x20000B2
TRANSPARENCY_40	0x2000099
TRANSPARENCY_50	0x200007F
TRANSPARENCY_60	0x2000066
TRANSPARENCY_70	0x200004C
TRANSPARENCY_80	0x2000032
TRANSPARENCY_90	0x2000019
TRANSPARENCY_BYBLOCK	0x1000000

6.8.6 Data Query

See also:

For usage of the query features see the tutorial: [Tutorial for getting data from DXF files](#)

Entity Query String

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

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

Entity Query

The entity query is a whitespace separated list of DXF entity names or the special name '*' . Where '*' means all DXF entities, exclude some entity types by appending their names with a preceding ! (e.g. all entities except LINE = '*' !LINE'). All DXF names have to be uppercase.

Attribute Query

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

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

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

- “==” equal “value”
- “!=” not equal “value”

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

Query Result

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

examples:

- LINE [text ? ".*"] : always empty, because the LINE entity has no text attribute.
- LINE CIRCLE[layer=="construction"] : all LINE and CIRCLE entities with layer == "construction"
- *[! (layer=="construction" & color<7)] : all entities except those with layer == "construction" and color < 7
- *[layer=="construction"]i, (ignore case) all entities with layer == "construction" | "Construction" | "ConStruction" ...

EntityQuery Class

`class ezdxf.query.EntityQuery`

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

first

First entity or `None`.

last

Last entity or `None`.

__len__() → int

Returns count of DXF entities.

__getitem__(item)

Returns DXFEntity at index `item`, supports negative indices and slicing.

__iter__() → Iterator[DXFEntity]

Returns iterable of DXFEntity objects.

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

Extent the `EntityQuery` container by entities matching an additional query.

remove(query: str = '*') → None

Remove all entities from `EntityQuery` container matching this additional query.

```
query (query: str = '*') → ezdxf.query.EntityQuery
```

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

raises: ParseException (pyparsing.py)

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

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

Parameters

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

The new() Function

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

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

See also:

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

Groupby Function

```
ezdxf.groupby.groupby (entities: Iterable[DXFEntity], dxattrib: str = "", key: KeyFunc = None) → Dict[Hashable, List[DXFEntity]]
```

Groups a sequence of DXF entities by a DXF attribute like 'layer', returns a dict with *dxattrib* values as key and a list of entities matching this *dxattrib*. A key function can be used to combine some DXF attributes (e.g. layer and color) and should return a hashable data type like a tuple of strings, integers or floats, key function example:

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

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

Result dict for *dxattrib* = 'layer' may look like this:

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

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

```
{  
    ('0', 1): [ ... list of entities ],  
    ('0', 3): [ ... ],
```

(continues on next page)

(continued from previous page)

```
('0', 7): [ ... ],
('ExampleLayer1', 1): [ ... ],
('ExampleLayer1', 2): [ ... ],
('ExampleLayer1', 5): [ ... ],
('ExampleLayer2', 7): [ ... ],
...
}
```

All entity containers (modelspace, paperspace layouts and blocks) and the `EntityQuery` object have a dedicated `groupby()` method.

Parameters

- **entities** – sequence of DXF entities to group by a DXF attribute or a `key` function
- **dxfattrib** – grouping DXF attribute like 'layer'
- **key** – key function, which accepts a `DXFEntity` as argument and returns a hashable grouping key or `None` to ignore this entity

6.8.7 Math Utilities

Utility functions and classes located in module `ezdxf.math`.

Functions

`ezdxf.math.closest_point(base: Vertex, points: Iterable[Vertex]) → Vec3`

Returns closest point to `base`.

Parameters

- **base** – base point as `Vec3` compatible object
- **points** – iterable of points as `Vec3` compatible object

`ezdxf.math.uniform_knot_vector(count: int, order: int, normalize=False) → List[float]`

Returns an uniform knot vector for a B-spline of `order` and `count` control points.

`order = degree + 1`

Parameters

- **count** – count of control points
- **order** – spline order
- **normalize** – normalize values in range [0, 1] if `True`

`ezdxf.math.open_uniform_knot_vector(count: int, order: int, normalize=False) → List[float]`

Returns an open (clamped) uniform knot vector for a B-spline of `order` and `count` control points.

`order = degree + 1`

Parameters

- **count** – count of control points
- **order** – spline order
- **normalize** – normalize values in range [0, 1] if `True`

`ezdxf.math.required_knot_values(count: int, order: int) → int`

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

Parameters

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

Relationship:

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

- $k = p + 1$
- $2 \leq k \leq n + 1$

`ezdxf.math.xround(value: float, rounding: float = 0.0) → float`

Extended rounding function, argument *rounding* defines the rounding limit:

0	remove fraction
0.1	round next to x.1, x.2, ... x.0
0.25	round next to x.25, x.50, x.75 or x.00
0.5	round next to x.5 or x.0
1.0	round to a multiple of 1: remove fraction
2.0	round to a multiple of 2: xxx2, xxx4, xxx6 ...
5.0	round to a multiple of 5: xxx5 or xxx0
10.0	round to a multiple of 10: xx10, xx20, ...

Parameters

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

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

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

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

`ezdxf.math.area(vertices: Iterable[Vertex]) → float`

Returns the area of a polygon, returns the projected area in the xy-plane for 3D vertices.

`ezdxf.math.arc_angle_span_deg(start: float, end: float) → float`

Returns the counter clockwise angle span from *start* to *end* in degrees.

Returns the angle span in the range of [0, 360], 360 is a full circle. Full circle handling is a special case, because normalization of angles which describe a full circle would return 0 if treated as regular angles. e.g. (0, 360) → 360, (0, -360) → 360, (180, -180) → 360. Input angles with the same value always return 0 by definition: (0, 0) → 0, (-180, -180) → 0, (360, 360) → 0.

`ezdxf.math.arc_angle_span_rad(start: float, end: float) → float`

Returns the counter clockwise angle span from *start* to *end* in radians.

Returns the angle span in the range of [0, 2π], 2π is a full circle. Full circle handling is a special case, because normalization of angles which describe a full circle would return 0 if treated as regular angles. e.g. (0, 2π) → 2π , (0, - 2π) → 2π , (π , - π) → 2π . Input angles with the same value always return 0 by definition: (0, 0) → 0, (- π , - π) → 0, (2π , 2π) → 0.

`ezdxf.math.arc_segment_count(radius: float, angle: float, sagitta: float) → int`

Returns the count of required segments for the approximation of an arc for a given maximum *sagitta*.

Parameters

- **radius** – arc radius
- **angle** – angle span of the arc in radians
- **sagitta** – max. distance from the center of an arc segment to the center of its chord

`ezdxf.math.arc_chord_length(radius: float, sagitta: float) → float`

Returns the chord length for an arc defined by `radius` and the `sagitta`.

Parameters

- **radius** – arc radius
- **sagitta** – distance from the center of the arc to the center of its base

`ezdxf.math.ellipse_param_span(start_param: float, end_param: float) → float`

Returns the counter clockwise params span of an elliptic arc from start- to end param.

Returns the param span in the range $[0, 2\pi]$, 2π is a full ellipse. Full ellipse handling is a special case, because normalization of params which describe a full ellipse would return 0 if treated as regular params. e.g. $(0, 2\pi) \rightarrow 2\pi$, $(0, -2\pi) \rightarrow 2\pi$, $(\pi, -\pi) \rightarrow 2\pi$. Input params with the same value always return 0 by definition: $(0, 0) \rightarrow 0$, $(-\pi, -\pi) \rightarrow 0$, $(2\pi, 2\pi) \rightarrow 0$.

Alias to function: `ezdxf.math.arc_angle_span_rad()`

`ezdxf.math.has_matrix_2d_stretching(m: Matrix44) → bool`

Returns True if matrix `m` performs a non-uniform xy-scaling. Uniform scaling is not stretching in this context.

Does not check if the target system is a cartesian coordinate system, use the `Matrix44` property `is_cartesian` for that.

`ezdxf.math.has_matrix_3d_stretching(m: Matrix44) → bool`

Returns True if matrix `m` performs a non-uniform xyz-scaling. Uniform scaling is not stretching in this context.

Does not check if the target system is a cartesian coordinate system, use the `Matrix44` property `is_cartesian` for that.

Bulge Related Functions

See also:

Description of the [Bulge value](#).

`ezdxf.math.bulge_center(start_point: Vertex, end_point: Vertex, bulge: float) → Vec2`

Returns center of arc described by the given bulge parameters.

Based on Bulge Center by Lee Mac.

Parameters

- **start_point** – start point as `Vec2` compatible object
- **end_point** – end point as `Vec2` compatible object
- **bulge** – bulge value as float

`ezdxf.math.bulge_radius(start_point: Vertex, end_point: Vertex, bulge: float) → float`

Returns radius of arc defined by the given bulge parameters.

Based on Bulge Radius by Lee Mac

Parameters

- **start_point** – start point as `Vec2` compatible object
- **end_point** – end point as `Vec2` compatible object
- **bulge** – bulge value

`ezdxf.math.arc_to_bulge(center: Vertex, start_angle: float, end_angle: float, radius: float) → Tuple[Vec2, Vec2, float]`

Returns bulge parameters from arc parameters.

Parameters

- **center** – circle center point as `Vec2` compatible object
- **start_angle** – start angle in radians
- **end_angle** – end angle in radians
- **radius** – circle radius

Returns (`start_point`, `end_point`, `bulge`)

Return type tuple

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

Returns arc parameters from bulge parameters.

The arcs defined by bulge values of `LWPolyline` and 2D `Polyline` entities start at the vertex which includes the bulge value and ends at the following vertex.

Based on Bulge to Arc by [Lee Mac](#).

Parameters

- **start_point** – start vertex as `Vec2` compatible object
- **end_point** – end vertex as `Vec2` compatible object
- **bulge** – bulge value

Returns (`center`, `start_angle`, `end_angle`, `radius`)

Return type Tuple

`ezdxf.math.bulge_3_points(start_point: Vertex, end_point: Vertex, point: Vertex) → float`

Returns bulge value defined by three points.

Based on 3-Points to Bulge by [Lee Mac](#).

Parameters

- **start_point** – start point as `Vec2` compatible object
- **end_point** – end point as `Vec2` compatible object
- **point** – arbitrary point as `Vec2` compatible object

2D Functions

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

Returns the normal distance from `point` to 2D line defined by `start`- and `end` point.

`ezdxf.math.point_to_line_relation(point: Vec2, start: Vec2, end: Vec2, abs_tol=1e-10) → int`

Returns `-1` if `point` is left `line`, `+1` if `point` is right of `line` and `0` if `point` is on the `line`. The `line` is defined by two vertices given as arguments `start` and `end`.

Parameters

- **point** – 2D point to test as `Vec2`
- **start** – line definition point as `Vec2`
- **end** – line definition point as `Vec2`
- **abs_tol** – tolerance for minimum distance to line

`ezdxf.math.is_point_on_line_2d(point: Vec2, start: Vec2, end: Vec2, ray=True, abs_tol=1e-10)` → bool

Returns True if *point* is on *line*.

Parameters

- **point** – 2D point to test as `Vec2`
- **start** – line definition point as `Vec2`
- **end** – line definition point as `Vec2`
- **ray** – if True point has to be on the infinite ray, if False point has to be on the line segment
- **abs_tol** – tolerance for on line test

`ezdxf.math.is_point_left_of_line(point: Vec2, start: Vec2, end: Vec2, colinear=False)` → bool

Returns True if *point* is “left of line” defined by *start*- and *end* point, a colinear point is also “left of line” if argument *colinear* is True.

Parameters

- **point** – 2D point to test as `Vec2`
- **start** – line definition point as `Vec2`
- **end** – line definition point as `Vec2`
- **colinear** – a colinear point is also “left of line” if True

`ezdxf.math.is_point_in_polygon_2d(point: Vec2, polygon: Iterable[Vec2], abs_tol=1e-10)` → int

Test if *point* is inside *polygon*.

Parameters

- **point** – 2D point to test as `Vec2`
- **polygon** – iterable of 2D points as `Vec2`
- **abs_tol** – tolerance for distance check

Returns +1 for inside, 0 for on boundary line, -1 for outside

`ezdxf.math.convex_hull_2d(points: Iterable[Vertex])` → List[`Vertex`]

Returns 2D convex hull for *points*.

Parameters **points** – iterable of points as `Vec3` compatible objects, z-axis is ignored

`ezdxf.math.intersection_line_line_2d(line1: Tuple[Vec2, Vec2], line2: Tuple[Vec2, Vec2], virtual=True, abs_tol=1e-10)` → Optional[`Vec2`]

Compute the intersection of two lines in the xy-plane.

Parameters

- **line1** – start- and end point of first line to test e.g. ((x1, y1), (x2, y2)).
- **line2** – start- and end point of second line to test e.g. ((x3, y3), (x4, y4)).

- **virtual** – True returns any intersection point, False returns only real intersection points.
- **abs_tol** – tolerance for intersection test.

Returns None if there is no intersection point (parallel lines) or intersection point as `Vec2`

`ezdxf.math.intersect_polylines_2d(p1: Sequence[Vec2], p2: Sequence[Vec2], abs_tol=1e-10)`
→ `List[Vec2]`

Returns the intersection points for two polylines as list of `Vec2` objects, the list is empty if no intersection points exist. Does not return self intersection points of `p1` or `p2`. Duplicate intersection points are removed from the result list, but the list does not have a particular order! You can sort the result list by `result.sort()` to introduce an order.

Parameters

- **p1** – first polyline as sequence of `Vec2` objects
- **p2** – second polyline as sequence of `Vec2` objects
- **abs_tol** – absolute tolerance for comparisons

New in version 0.17.2.

`ezdxf.math.rytz_axis_construction(d1: Vec3, d2: Vec3)` → `Tuple[Vec3, Vec3, float]`

The Rytz's axis construction is a basic method of descriptive Geometry to find the axes, the semi-major axis and semi-minor axis, starting from two conjugated half-diameters.

Source: [Wikipedia](#)

Given conjugated diameter `d1` is the vector from center C to point P and the given conjugated diameter `d2` is the vector from center C to point Q. Center of ellipse is always (0, 0, 0). This algorithm works for 2D/3D vectors.

Parameters

- **d1** – conjugated semi-major axis as `Vec3`
- **d2** – conjugated semi-minor axis as `Vec3`

Returns Tuple of (major axis, minor axis, ratio)

`ezdxf.math.clip_polygon_2d(clip: Iterable[Vertex], subject: Iterable[Vertex], ccw_check: bool = True)` → `List[Vec2]`

Clip the `subject` polygon by the `convex` clipping polygon `clip`.

Implements the Sutherland–Hodgman algorithm for clipping polygons.

Parameters

- **clip** – the convex clipping polygon as iterable of vertices
- **subject** – the polygon to clip as a iterable of vertices
- **ccw_check** – check if the clipping polygon is in counter clockwise orientation if True, set to False if the ccw check is done by the caller

Returns the clipped subject as list of `Vec2`

New in version 0.16.

`ezdxf.math.offset_vertices_2d(vertices: Iterable[Vertex], offset: float, closed: bool = False)` → `Iterable[Vec2]`

Yields vertices of the offset line to the shape defined by `vertices`. The source shape consist of straight segments and is located in the xy-plane, the z-axis of input vertices is ignored. Takes closed shapes into account if argument `closed` is True, which yields intersection of first and last offset segment as first vertex for a closed

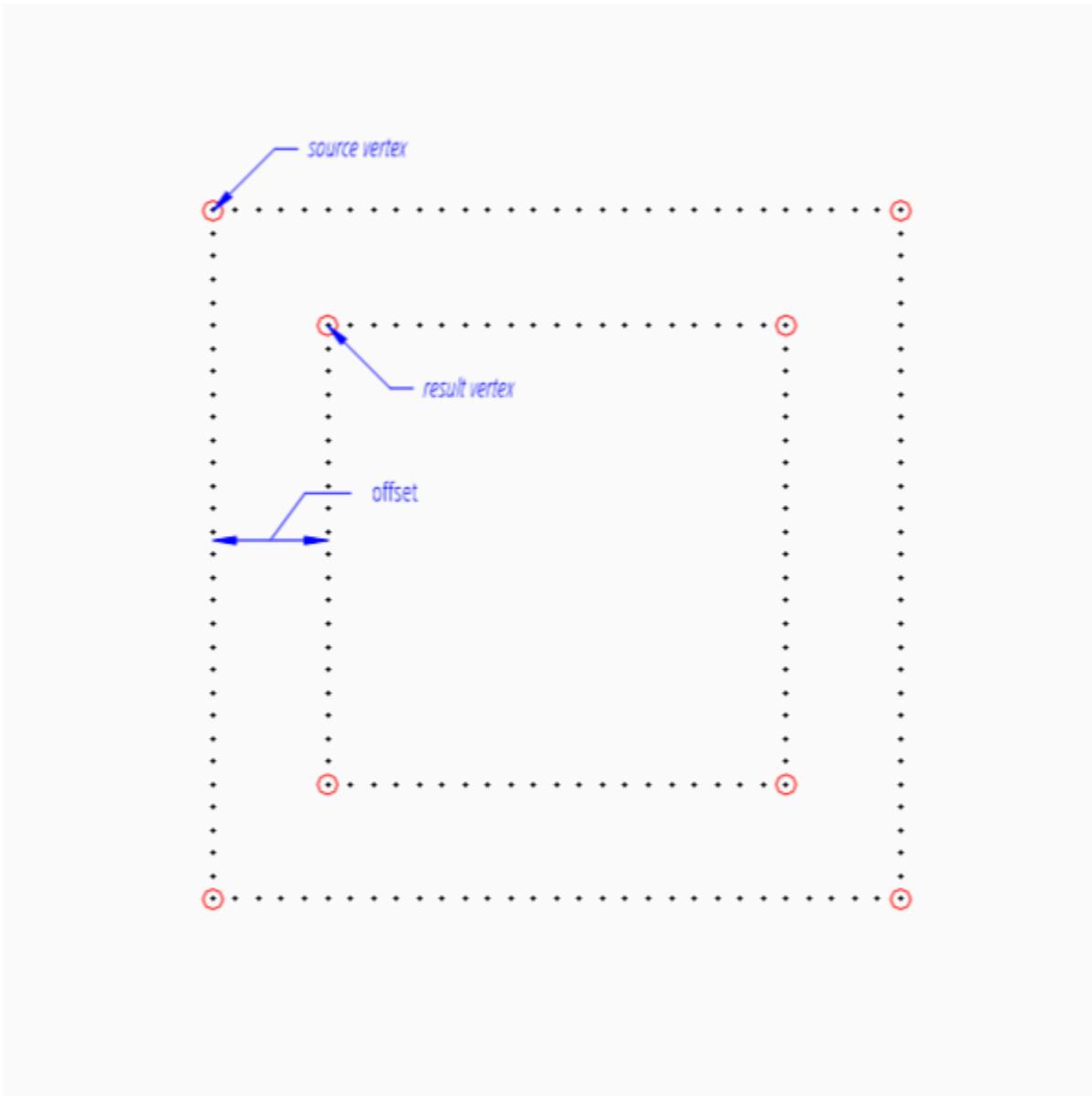
shape. For closed shapes the first and last vertex can be equal, else an implicit closing segment from last to first vertex is added. A shape with equal first and last vertex is not handled automatically as closed shape.

Warning: Adjacent collinear segments in *opposite* directions, same as a turn by 180 degree (U-turn), leads to unexpected results.

Parameters

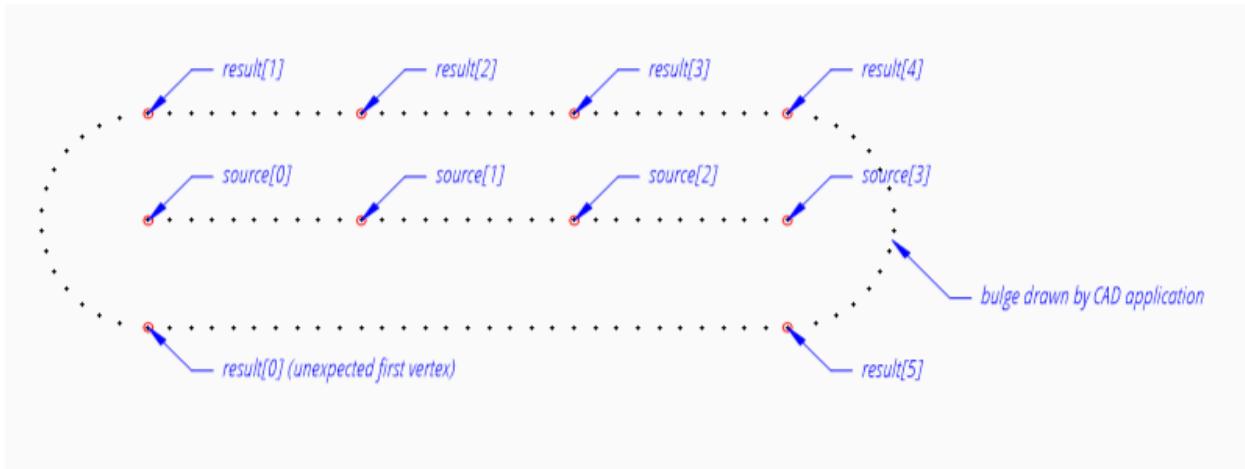
- **vertices** – source shape defined by vertices
- **offset** – line offset perpendicular to direction of shape segments defined by vertices order, offset > 0 is ‘left’ of line segment, offset < 0 is ‘right’ of line segment
- **closed** – True to handle as closed shape

```
source = [(0, 0), (3, 0), (3, 3), (0, 3)]
result = list(offset_vertices_2d(source, offset=0.5, closed=True))
```



Example for a closed collinear shape, which creates 2 additional vertices and the first one has an unexpected location:

```
source = [(0, 0), (0, 1), (0, 2), (0, 3)]
result = list(offset_vertices_2d(source, offset=0.5, closed=True))
```



3D Functions

See also:

The free online book [3D Math Primer for Graphics and Game Development](#) is a very good resource for learning vector math and other graphic related topics, it is easy to read for beginners and especially targeted to programmers.

`ezdxf.math.basic_transformation (move: Vertex = (0, 0, 0), scale: Vertex = (1, 1, 1), z_rotation: float = 0) → Matrix44`

Returns a combined transformation matrix for translation, scaling and rotation about the z-axis.

Parameters

- **move** – translation vector
- **scale** – x-, y- and z-axis scaling as float triplet, e.g. (2, 2, 1)
- **z_rotation** – rotation angle about the z-axis in radians

`ezdxf.math.normal_vector_3p (a: Vec3, b: Vec3, c: Vec3) → Vec3`

Returns normal vector for 3 points, which is the normalized cross product for: $a \rightarrow b \times a \rightarrow c$.

`ezdxf.math.linear_vertex_spacing (start: Vec3, end: Vec3, count: int) → List[Vec3]`

Returns *count* evenly spaced vertices from *start* to *end*.

`ezdxf.math.best_fit_normal (vertices: Iterable[Vertex]) → Vec3`

Returns the “best fit” normal for a plane defined by three or more vertices. This function tolerates imperfect plane vertices. Safe function to detect the extrusion vector of flat arbitrary polygons.

`ezdxf.math.is_planar_face (face: Sequence[Vec3], abs_tol=1e-9) → bool`

Returns True if sequence of vectors is a planar face.

Parameters

- **face** – sequence of `Vec3` objects
- **abs_tol** – tolerance for normals check

`ezdxf.math.subdivide_face (face: Sequence[Union[Vec3, Vec2]], quads=True) → Iterable[List[Vec3]]`

Yields new subdivided faces. Creates new faces from subdivided edges and the face midpoint by linear interpolation.

Parameters

- **face** – a sequence of vertices, `Vec2` and `Vec3` objects supported.

- **quads** – create quad faces if True else create triangles

```
ezdxf.math.subdivide_ngons(faces: Iterable[Sequence[Union[Vec3, Vec2]]]) → Iterable[List[Vec3]]
Yields only triangles or quad faces, subdivides ngons into triangles.
```

Parameters **faces** – iterable of faces as sequence of `Vec2` and `Vec3` objects

```
ezdxf.math.distance_point_line_3d(point: Vec3, start: Vec3, end: Vec3) → float
Returns the normal distance from point to 3D line defined by start- and end point.
```

```
ezdxf.math.intersection_ray_ray_3d(ray1: Tuple[Vec3, Vec3], ray2: Tuple[Vec3, Vec3],
                                     abs_tol=1e-10) → Sequence[Vec3]
Calculate intersection of two 3D rays, returns a 0-tuple for parallel rays, a 1-tuple for intersecting rays and a 2-tuple for not intersecting and not parallel rays with points of closest approach on each ray.
```

Parameters

- **ray1** – first ray as tuple of two points as `Vec3` objects
- **ray2** – second ray as tuple of two points as `Vec3` objects
- **abs_tol** – absolute tolerance for comparisons

```
ezdxf.math.intersection_line_line_3d(line1: Tuple[Vec3, Vec3], line2: Tuple[Vec3, Vec3], virtual=True,
                                         abs_tol=1e-10) → Optional[Vec3]
```

Returns the intersection point of two 3D lines, returns `None` if lines do not intersect.

Parameters

- **line1** – first line as tuple of two points as `Vec3` objects
- **line2** – second line as tuple of two points as `Vec3` objects
- **virtual** – `True` returns any intersection point, `False` returns only real intersection points
- **abs_tol** – absolute tolerance for comparisons

New in version 0.17.2.

```
ezdxf.math.intersect_polylines_3d(p1: Sequence[Vec3], p2: Sequence[Vec3], abs_tol=1e-10)
                                         → List[Vec3]
```

Returns the intersection points for two polylines as list of `Vec3` objects, the list is empty if no intersection points exist. Does not return self intersection points of *p1* or *p2*. Duplicate intersection points are removed from the result list, but the list does not have a particular order! You can sort the result list by `result.sort()` to introduce an order.

Parameters

- **p1** – first polyline as sequence of `Vec3` objects
- **p2** – second polyline as sequence of `Vec3` objects
- **abs_tol** – absolute tolerance for comparisons

New in version 0.17.2.

```
ezdxf.math.estimate_tangents(points: List[Vec3], method: str = '5-points', normalize = True) → List[Vec3]
```

Estimate tangents for curve defined by given fit points. Calculated tangents are normalized (unit-vectors).

Available tangent estimation methods:

- “3-points”: 3 point interpolation
- “5-points”: 5 point interpolation

- “bezier”: tangents from an interpolated cubic bezier curve
- “diff”: finite difference

Parameters

- **points** – start-, end- and passing points of curve
- **method** – tangent estimation method
- **normalize** – normalize tangents if True

Returns tangents as list of `Vec3` objects

```
ezdxf.math.estimate_end_tangent_magnitude(points: List[Vec3], method: str = 'chord') →
List[Vec3]
```

Estimate tangent magnitude of start- and end tangents.

Available estimation methods:

- “chord”: total chord length, curve approximation by straight segments
- “arc”: total arc length, curve approximation by arcs
- “bezier-n”: total length from cubic bezier curve approximation, n segments per section

Parameters

- **points** – start-, end- and passing points of curve
- **method** – tangent magnitude estimation method

```
ezdxf.math.fit_points_to_cad_cv(fit_points: Iterable[Vertex], tangents: Iterable[Vertex] = None,
estimate: str = '5-p') → BSpline
```

Returns a cubic `BSpline` from fit points as close as possible to common CAD applications like BricsCAD.

There exist infinite numerical correct solution for this setup, but some facts are known:

- Global curve interpolation with start- and end derivatives, e.g. 6 fit points creates 8 control vertices in BricsCAD
- Degree of B-spline is always 3, the stored degree is ignored, this is only valid for B-splines defined by fit points
- Knot parametrization method is “chord”
- Knot distribution is “natural”

The last missing parameter is the start- and end tangents estimation method used by BricsCAD, if these tangents are stored in the DXF file provide them as argument `tangents` as 2-tuple (start, end) and the interpolated control vertices will match the BricsCAD calculation, except for floating point imprecision.

If the end tangents are not given, the start- and ent tangent directions will be estimated. The argument `estimate` lets choose from different estimation methods (first 3 letters are significant):

- “3-points”: 3 point interpolation
- “5-points”: 5 point interpolation
- “bezier”: tangents from an interpolated cubic bezier curve
- “diff”: finite difference

The estimation method “5-p” yields the closest match to the BricsCAD rendering, but sometimes “bez” creates a better result.

If I figure out how BricsCAD estimates the end tangents directions, the argument *estimate* gets an additional value for that case. The existing estimation methods will perform the same way as now, except for bug fixes. But the default value may change, therefore set argument *estimate* to specific value to always get the same result in the future.

Parameters

- **fit_points** – points the spline is passing through
- **tangents** – start- and end tangent, default is autodetect
- **estimate** – tangent direction estimation method

Changed in version 0.16: removed unused arguments *degree* and *method*

`ezdxf.math.fit_points_to_cubic_bezier(fit_points: Iterable[Vertex]) → BSpline`

Returns a cubic *BSpline* from fit points **without** end tangents.

This function uses the cubic Bézier interpolation to create multiple Bézier curves and combine them into a single B-spline, this works for short simple splines better than the `fit_points_to_cad_cv()`, but is worse for longer and more complex splines.

Parameters **fit_points** – points the spline is passing through

New in version 0.16.

`ezdxf.math.global_bspline_interpolation(fit_points: Iterable[Vertex], degree: int = 3, tangents: Iterable[Vertex] = None, method: str = 'chord') → BSpline`

B-spline interpolation by the [Global Curve Interpolation](#). Given are the fit points and the degree of the B-spline. The function provides 3 methods for generating the parameter vector t:

- “uniform”: creates a uniform t vector, from 0 to 1 evenly spaced, see [uniform](#) method
- “chord”, “distance”: creates a t vector with values proportional to the fit point distances, see [chord length](#) method
- “centripetal”, “sqrt_chord”: creates a t vector with values proportional to the fit point $\text{sqrt}(\text{distances})$, see [centripetal](#) method
- “arc”: creates a t vector with values proportional to the arc length between fit points.

It is possible to constraint the curve by tangents, by start- and end tangent if only two tangents are given or by one tangent for each fit point.

If tangents are given, they represent 1st derivatives and should be scaled if they are unit vectors, if only start- and end tangents given the function `estimate_end_tangent_magnitude()` helps with an educated guess, if all tangents are given, scaling by chord length is a reasonable choice (Piegl & Tiller).

Parameters

- **fit_points** – fit points of B-spline, as list of [Vec3](#) compatible objects
- **tangents** – if only two vectors are given, take the first and the last vector as start- and end tangent constraints or if for all fit points a tangent is given use all tangents as interpolation constraints (optional)
- **degree** – degree of B-spline
- **method** – calculation method for parameter vector t

Returns *BSpline*

```
ezdxf.math.local_cubic_bspline_interpolation(fit_points: Iterable[Vertex], method: str
                                              = '5-points', tangents: Iterable[Vertex] =
                                              None) → BSpline
```

B-spline interpolation by ‘Local Cubic Curve Interpolation’, which creates B-spline from fit points and estimated tangent direction at start-, end- and passing points.

Source: Piegl & Tiller: “The NURBS Book” - chapter 9.3.4

Available tangent estimation methods:

- “3-points”: 3 point interpolation
- “5-points”: 5 point interpolation
- “bezier”: cubic bezier curve interpolation
- “diff”: finite difference

or pass pre-calculated tangents, which overrides tangent estimation.

Parameters

- **fit_points** – all B-spline fit points as `Vec3` compatible objects
- **method** – tangent estimation method
- **tangents** – tangents as `Vec3` compatible objects (optional)

Returns `BSpline`

```
ezdxf.math.rational_bspline_from_arc(center: Vec3 = (0, 0), radius: float = 1, start_angle: float
                                       = 0, end_angle: float = 360, segments: int = 1) →
BSpline
```

Returns a rational B-splines for a circular 2D arc.

Parameters

- **center** – circle center as `Vec3` compatible object
- **radius** – circle radius
- **start_angle** – start angle in degrees
- **end_angle** – end angle in degrees
- **segments** – count of spline segments, at least one segment for each quarter (90 deg), default is 1, for as few as needed.

```
ezdxf.math.rational_bspline_from_ellipse(ellipse: ConstructionEllipse, segments: int = 1)
                                         → BSpline
```

Returns a rational B-splines for an elliptic arc.

Parameters

- **ellipse** – ellipse parameters as `ConstructionEllipse` object
- **segments** – count of spline segments, at least one segment for each quarter ($\pi/2$), default is 1, for as few as needed.

```
ezdxf.math.open_uniform_bspline(control_points: Iterable[Vertex], order: int = 4, weights: Iterable[float] = None) → BSpline
```

Creates an open uniform (periodic) B-spline curve (open curve).

This is an unclamped curve, which means the curve passes none of the control points.

Parameters

- **control_points** – iterable of control points as `Vec3` compatible objects

- **order** – spline order (degree + 1)
- **weights** – iterable of weight values

`ezdxf.math.closed_uniform_bspline(control_points: Iterable[Vertex], order: int = 4, weights: Iterable[float] = None) → BSpline`

Creates an closed uniform (periodic) B-spline curve (open curve).

This B-spline does not pass any of the control points.

Parameters

- **control_points** – iterable of control points as `Vec3` compatible objects
- **order** – spline order (degree + 1)
- **weights** – iterable of weight values

`ezdxf.math.cubic_bezier_from_arc(center: Vec3 = (0, 0), radius: float = 1, start_angle: float = 0, end_angle: float = 360, segments: int = 1) → Iterable[Bezier4P]`

Returns an approximation for a circular 2D arc by multiple cubic Bézier-curves.

Parameters

- **center** – circle center as `Vec3` compatible object
- **radius** – circle radius
- **start_angle** – start angle in degrees
- **end_angle** – end angle in degrees
- **segments** – count of Bézier-curve segments, at least one segment for each quarter (90 deg), 1 for as few as possible.

`ezdxf.math.cubic_bezier_from_ellipse(ellipse: ConstructionEllipse, segments: int = 1) → Iterable[Bezier4P]`

Returns an approximation for an elliptic arc by multiple cubic Bézier-curves.

Parameters

- **ellipse** – ellipse parameters as `ConstructionEllipse` object
- **segments** – count of Bézier-curve segments, at least one segment for each quarter ($\pi/2$), 1 for as few as possible.

`ezdxf.math.cubic_bezier_from_3p(p1: Vertex, p2: Vertex, p3: Vertex) → Bezier4P`

Returns a cubic Bézier curve `Bezier4P` from three points. The curve starts at *p1*, goes through *p2* and ends at *p3*. (source: pomax-2)

New in version 0.17.2.

`ezdxf.math.cubic_bezier_interpolation(points: Iterable[Vertex]) → List[Bezier4P]`

Returns an interpolation curve for given data *points* as multiple cubic Bézier-curves. Returns n-1 cubic Bézier-curves for n given data points, curve i goes from point[i] to point[i+1].

Parameters **points** – data points

`ezdxf.math.quadratic_to_cubic_bezier(bezier: Bezier3P) → Bezier4P`

Convert quadratic Bézier curves (`ezdxf.math.Bezier3P`) into cubic Bézier curves (`ezdxf.math.Bezier4P`).

`ezdxf.math.quadratic_bezier_from_3p(p1: Vertex, p2: Vertex, p3: Vertex) → Bezier3P`

Returns a quadratic Bézier curve `Bezier3P` from three points. The curve starts at *p1*, goes through *p2* and ends at *p3*. (source: pomax-2)

New in version 0.17.2.

`ezdxf.math.bezier_to_bspline(Iterable[Union[Bezier3P, Bezier4P]], segments int = 4) → BSpline`

Convert multiple quadratic or cubic Bézier curves into a single cubic B-spline (`ezdxf.math.BSpline`). For good results the curves must be lined up seamlessly, i.e. the starting point of the following curve must be the same as the end point of the previous curve. G1 continuity or better at the connection points of the Bézier curves is required to get best results.

`ezdxf.math.have_bezier_curves_g1_continuity(b1: AnyBezier, b2 AnyBezier, g1_tol: float = 1e-4) → bool`

Return `True` if the given adjacent bezier curves have G1 continuity.

`ezdxf.math.split_bezier(control_points: Sequence[AnyVec], t: float) → Tuple[List[AnyVec], List[AnyVec]]`

Split Bézier curves at parameter `t` by de Casteljau's algorithm (source: pomax-1). Returns the control points for two new Bézier curves of the same degree and type as the input curve.

Parameters

- `control_points` – of the Bézier curve as `Vec2` or `Vec3` objects. Requires 3 points for a quadratic curve, 4 points for a cubic curve , ...
- `t` – parameter where to split the curve in the range [0, 1]

New in version 0.17.2.

Transformation Classes

OCS Class

`class eздxf.math.OCS(extrusion: Vertex = Vec3(0.0, 0.0, 1.0))`

Establish an `OCS` for a given extrusion vector.

Parameters `extrusion` – extrusion vector.

ux

x-axis unit vector

uy

y-axis unit vector

uz

z-axis unit vector

`from_wcs(point: Vertex) → Vertex`

Returns OCS vector for WCS `point`.

`points_from_wcs(points: Iterable[Vertex]) → Iterable[Vertex]`

Returns iterable of OCS vectors from WCS `points`.

`to_wcs(point: Vertex) → Vertex`

Returns WCS vector for OCS `point`.

`points_to_wcs(points: Iterable[Vertex]) → Iterable[Vertex]`

Returns iterable of WCS vectors for OCS `points`.

`render_axis(layout: BaseLayout, length: float = 1, colors: RGB = (1, 3, 5)) → None`

Render axis as 3D lines into a `layout`.

UCS Class

class `ezdxf.math.UCS` (*origin: Vertex* = $(0, 0, 0)$, *ux: Vertex* = *None*, *uy: Vertex* = *None*, *uz: Vertex* = *None*)

Establish an user coordinate system (*UCS*). The UCS is defined by the origin and two unit vectors for the x-, y- or z-axis, all axis in *WCS*. The missing axis is the cross product of the given axis.

If x- and y-axis are *None*: *ux* = $(1, 0, 0)$, *uy* = $(0, 1, 0)$, *uz* = $(0, 0, 1)$.

Unit vectors don't have to be normalized, normalization is done at initialization, this is also the reason why scaling gets lost by copying or rotating.

Parameters

- **origin** – defines the UCS origin in world coordinates
- **ux** – defines the UCS x-axis as vector in *WCS*
- **uy** – defines the UCS y-axis as vector in *WCS*
- **uz** – defines the UCS z-axis as vector in *WCS*

ux

x-axis unit vector

uy

y-axis unit vector

uz

z-axis unit vector

is_cartesian

Returns True if cartesian coordinate system.

copy() → UCS

Returns a copy of this UCS.

to_wcs (*point: ezdxf.math._vector.Vec3*) → `ezdxf.math._vector.Vec3`

Returns WCS point for UCS *point*.

points_to_wcs (*points: Iterable[ezdxf.math._vector.Vec3]*) → `Iterable[ezdxf.math._vector.Vec3]`

Returns iterable of WCS vectors for UCS *points*.

direction_to_wcs (*vector: ezdxf.math._vector.Vec3*) → `ezdxf.math._vector.Vec3`

Returns WCS direction for UCS *vector* without origin adjustment.

from_wcs (*point: ezdxf.math._vector.Vec3*) → `ezdxf.math._vector.Vec3`

Returns UCS point for WCS *point*.

points_from_wcs (*points: Iterable[ezdxf.math._vector.Vec3]*) → `Iterable[ezdxf.math._vector.Vec3]`

Returns iterable of UCS vectors from WCS *points*.

direction_from_wcs (*vector: ezdxf.math._vector.Vec3*) → `ezdxf.math._vector.Vec3`

Returns UCS vector for WCS *vector* without origin adjustment.

to_octs (*point: ezdxf.math._vector.Vec3*) → `ezdxf.math._vector.Vec3`

Returns OCS vector for UCS *point*.

The *OCS* is defined by the z-axis of the *UCS*.

points_to_octs (*points: Iterable[ezdxf.math._vector.Vec3]*) → `Iterable[ezdxf.math._vector.Vec3]`

Returns iterable of OCS vectors for UCS *points*.

The *OCS* is defined by the z-axis of the *UCS*.

Parameters `points` – iterable of UCS vertices

`to_ocs_angle_deg(angle: float) → float`

Transforms `angle` from current UCS to the parent coordinate system (most likely the WCS) including the transformation to the OCS established by the extrusion vector `UCS.uz`.

Parameters `angle` – in UCS in degrees

`transform(m: Matrix44) → UCS`

General inplace transformation interface, returns `self` (floating interface).

Parameters `m` – 4x4 transformation matrix (`ezdxf.math.Matrix44`)

`rotate(axis: Vertex, angle:float) → UCS`

Returns a new rotated UCS, with the same origin as the source UCS. The rotation vector is located in the origin and has `WCS` coordinates e.g. (0, 0, 1) is the WCS z-axis as rotation vector.

Parameters

- `axis` – arbitrary rotation axis as vector in `WCS`
- `angle` – rotation angle in radians

`rotate_local_x(angle:float) → UCS`

Returns a new rotated UCS, rotation axis is the local x-axis.

Parameters `angle` – rotation angle in radians

`rotate_local_y(angle:float) → UCS`

Returns a new rotated UCS, rotation axis is the local y-axis.

Parameters `angle` – rotation angle in radians

`rotate_local_z(angle:float) → UCS`

Returns a new rotated UCS, rotation axis is the local z-axis.

Parameters `angle` – rotation angle in radians

`shift(delta: Vertex) → UCS`

Shifts current UCS by `delta` vector and returns `self`.

Parameters `delta` – shifting vector

`moveto(location: Vertex) → UCS`

Place current UCS at new origin `location` and returns `self`.

Parameters `location` – new origin in WCS

`static from_x_axis_and_point_in_xy(origin: Vertex, axis: Vertex, point: Vertex) → UCS`

Returns an new `UCS` defined by the origin, the x-axis vector and an arbitrary point in the xy-plane.

Parameters

- `origin` – UCS origin as (x, y, z) tuple in `WCS`
- `axis` – x-axis vector as (x, y, z) tuple in `WCS`
- `point` – arbitrary point unlike the origin in the xy-plane as (x, y, z) tuple in `WCS`

`static from_x_axis_and_point_in_xz(origin: Vertex, axis: Vertex, point: Vertex) → UCS`

Returns an new `UCS` defined by the origin, the x-axis vector and an arbitrary point in the xz-plane.

Parameters

- `origin` – UCS origin as (x, y, z) tuple in `WCS`
- `axis` – x-axis vector as (x, y, z) tuple in `WCS`

- **point** – arbitrary point unlike the origin in the xz-plane as (x, y, z) tuple in [WCS](#)

static from_y_axis_and_point_in_xy (*origin*: Vertex, *axis*: Vertex, *point*: Vertex) → UCS
Returns an new [UCS](#) defined by the origin, the y-axis vector and an arbitrary point in the xy-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in [WCS](#)
- **axis** – y-axis vector as (x, y, z) tuple in [WCS](#)
- **point** – arbitrary point unlike the origin in the xy-plane as (x, y, z) tuple in [WCS](#)

static from_y_axis_and_point_in_yz (*origin*: Vertex, *axis*: Vertex, *point*: Vertex) → UCS
Returns an new [UCS](#) defined by the origin, the y-axis vector and an arbitrary point in the yz-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in [WCS](#)
- **axis** – y-axis vector as (x, y, z) tuple in [WCS](#)
- **point** – arbitrary point unlike the origin in the yz-plane as (x, y, z) tuple in [WCS](#)

static from_z_axis_and_point_in_xz (*origin*: Vertex, *axis*: Vertex, *point*: Vertex) → UCS
Returns an new [UCS](#) defined by the origin, the z-axis vector and an arbitrary point in the xz-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in [WCS](#)
- **axis** – z-axis vector as (x, y, z) tuple in [WCS](#)
- **point** – arbitrary point unlike the origin in the xz-plane as (x, y, z) tuple in [WCS](#)

static from_z_axis_and_point_in_yz (*origin*: Vertex, *axis*: Vertex, *point*: Vertex) → UCS
Returns an new [UCS](#) defined by the origin, the z-axis vector and an arbitrary point in the yz-plane.

Parameters

- **origin** – UCS origin as (x, y, z) tuple in [WCS](#)
- **axis** – z-axis vector as (x, y, z) tuple in [WCS](#)
- **point** – arbitrary point unlike the origin in the yz-plane as (x, y, z) tuple in [WCS](#)

render_axis (*layout*: BaseLayout, *length*: float = 1, *colors*: Tuple[int, int, int] = (1, 3, 5))
Render axis as 3D lines into a *layout*.

Matrix44

class eздxf.math.Matrix44 (*args)

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

The utility functions for constructing transformations and transforming vectors and points assumes that vectors are stored as row vectors, meaning when multiplied, transformations are applied left to right (e.g. vAB transforms v by A then by B).

Matrix44 initialization:

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

__repr__() → str
 Returns the representation string of the matrix: Matrix44((col0, col1, col2, col3), (..), (...), (...))

get_row(row: int) → Tuple[float, ...]
 Get row as list of four float values.

Parameters **row** – row index [0 .. 3]

set_row(row: int, values: Sequence[float]) → None
 Sets the values in a row.

Parameters

- **row** – row index [0 .. 3]
- **values** – iterable of four row values

get_col(col: int) → Tuple[float, ...]
 Returns a column as a tuple of four floats.

Parameters **col** – column index [0 .. 3]

set_col(col: int, values: Sequence[float]) → None
 Sets the values in a column.

Parameters

- **col** – column index [0 .. 3]
- **values** – iterable of four column values

copy() → Matrix44
 Returns a copy of same type.

__copy__() → Matrix44
 Returns a copy of same type.

classmethod scale(sx: float, sy: float = None, sz: float = None) → Matrix44
 Returns a scaling transformation matrix. If sy is None, sy = sx, and if sz is None sz = sx.

classmethod translate(dx: float, dy: float, dz: float) → Matrix44
 Returns a translation matrix for translation vector (dx, dy, dz).

classmethod x_rotate(angle: float) → Matrix44
 Returns a rotation matrix about the x-axis.

Parameters **angle** – rotation angle in radians

classmethod y_rotate(angle: float) → Matrix44
 Returns a rotation matrix about the y-axis.

Parameters **angle** – rotation angle in radians

classmethod z_rotate(angle: float) → Matrix44
 Returns a rotation matrix about the z-axis.

Parameters **angle** – rotation angle in radians

classmethod axis_rotate(axis: Vertex, angle: float) → Matrix44
 Returns a rotation matrix about an arbitrary axis.

Parameters

- **axis** – rotation axis as (x, y, z) tuple or `Vec3` object
- **angle** – rotation angle in radians

classmethod xyz_rotate(*angle_x: float, angle_y: float, angle_z: float*) → Matrix44

Returns a rotation matrix for rotation about each axis.

Parameters

- **angle_x** – rotation angle about x-axis in radians
- **angle_y** – rotation angle about y-axis in radians
- **angle_z** – rotation angle about z-axis in radians

classmethod shear_xy(*angle_x: float, angle_y: float*) → Matrix44

Returns a translation matrix for shear mapping (visually similar to slanting) in the xy-plane.

Parameters

- **angle_x** – slanting angle in x direction in radians
- **angle_y** – slanting angle in y direction in radians

classmethod perspective_projection(*left: float, right: float, top: float, bottom: float, near: float, far: float*) → Matrix44

Returns a matrix for a 2D projection.

Parameters

- **left** – Coordinate of left of screen
- **right** – Coordinate of right of screen
- **top** – Coordinate of the top of the screen
- **bottom** – Coordinate of the bottom of the screen
- **near** – Coordinate of the near clipping plane
- **far** – Coordinate of the far clipping plane

classmethod perspective_projection_fov(*fov: float, aspect: float, near: float, far: float*) → Matrix44

Returns a matrix for a 2D projection.

Parameters

- **fov** – The field of view (in radians)
- **aspect** – The aspect ratio of the screen (width / height)
- **near** – Coordinate of the near clipping plane
- **far** – Coordinate of the far clipping plane

static chain(**matrices: Iterable[Matrix44]*) → Matrix44

Compose a transformation matrix from one or more *matrices*.

static ucs(*ux: Vertex, uy: Vertex, uz: Vertex*) → Matrix44

Returns a matrix for coordinate transformation from WCS to UCS. For transformation from UCS to WCS, transpose the returned matrix.

Parameters

- **ux** – x-axis for UCS as unit vector
- **uy** – y-axis for UCS as unit vector
- **uz** – z-axis for UCS as unit vector
- **origin** – UCS origin as location vector

`__hash__()`
Return hash(self).

`__getitem__(index: Tuple[int, int])`
Get (row, column) element.

`__setitem__(index: Tuple[int, int], value: float)`
Set (row, column) element.

`__iter__()` → Iterator[float]
Iterates over all matrix values.

`rows()` → Iterable[Tuple[float, ...]]
Iterate over rows as 4-tuples.

`columns()` → Iterable[Tuple[float, ...]]
Iterate over columns as 4-tuples.

`mul__(other: Matrix44) → Matrix44`
Returns a new matrix as result of the matrix multiplication with another matrix.

`imul__(other: Matrix44) → Matrix44`
Inplace multiplication with another matrix.

`transform(vector: Vertex) → ezdxf.math._vector.Vec3`
Returns a transformed vertex.

`transform_direction(vector: Vertex, normalize=False) → ezdxf.math._vector.Vec3`
Returns a transformed direction vector without translation.

`transform_vertices(vectors: Iterable[Vertex]) → Iterable[ezdxf.math._vector.Vec3]`
Returns an iterable of transformed vertices.

`transform_directions(vectors: Iterable[Vertex], normalize=False) → Iterable[ezdxf.math._vector.Vec3]`
Returns an iterable of transformed direction vectors without translation.

`transpose() → None`
Swaps the rows for columns inplace.

`determinant() → float`
Returns determinant.

`inverse() → None`
Calculates the inverse of the matrix.
Raises ZeroDivisionError – if matrix has no inverse.

Construction Tools

Vec3

```
class ezdxf.math.Vec3(*args)
```

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

```
v1 = Vec3(1, 2, 3)
v2 = v1
v2.z = 7 # this is not possible, raises AttributeError
v2 = Vec3(v2.x, v2.y, 7) # this creates a new Vec3() object
assert v1.z == 3 # and v1 remains unchanged
```

`Vec3` initialization:

- `Vec3()`, returns `Vec3(0, 0, 0)`
- `Vec3((x, y))`, returns `Vec3(x, y, 0)`
- `Vec3((x, y, z))`, returns `Vec3(x, y, z)`
- `Vec3(x, y)`, returns `Vec3(x, y, 0)`
- `Vec3(x, y, z)`, returns `Vec3(x, y, z)`

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

```
v = Vec3(1, 2, 3)
v + (1, 2, 3) == Vec3(2, 4, 6)
(1, 2, 3) + v == Vec3(2, 4, 6)
v - (1, 2, 3) == Vec3(0, 0, 0)
(1, 2, 3) - v == Vec3(0, 0, 0)
v * 3 == Vec3(3, 6, 9)
3 * v == Vec3(3, 6, 9)
Vec3(3, 6, 9) / 3 == Vec3(1, 2, 3)
-Vec3(1, 2, 3) == (-1, -2, -3)
```

Comparison between vectors and vectors or tuples is supported:

```
Vec3(1, 2, 3) < Vec3(2, 2, 2)
(1, 2, 3) < tuple(Vec3(2, 2, 2)) # conversion necessary
Vec3(1, 2, 3) == (1, 2, 3)

bool(Vec3(1, 2, 3)) is True
bool(Vec3(0, 0, 0)) is False
```

x

x-axis value

y

y-axis value

z

z-axis value

xy

`Vec3` as `(x, y, 0)`, projected on the xy-plane.

xyz

`Vec3` as `(x, y, z)` tuple.

vec2

Real 2D vector as `Vec2` object.

magnitude

Length of vector.

magnitude_xy

Length of vector in the xy-plane.

magnitude_square

Square length of vector.

is_null

`Vec3(0, 0, 0)`. Has a fixed absolute testing tolerance of 1e-12!

Type `True` if all components are close to zero

angle

Angle between vector and x-axis in the xy-plane in radians.

angle_deg

Returns angle of vector and x-axis in the xy-plane in degrees.

spatial_angle

Spatial angle between vector and x-axis in radians.

spatial_angle_deg

Spatial angle between vector and x-axis in degrees.

__str__ () → str

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

__repr__ () → str

Return 'Vec3(x, y, z)' as string.

__len__ () → int

Returns always 3.

__hash__ () → int

Returns hash value of vector, enables the usage of vector as key in set and dict.

copy () → Vec3

Returns a copy of vector as `Vec3` object.

__copy__ () → Vec3

Returns a copy of vector as `Vec3` object.

__deepcopy__ (memodict: dict) → Vec3

`copy.deepcopy()` support.

__getitem__ (index: int) → float

Support for indexing:

- v[0] is v.x

- v[1] is v.y

- v[2] is v.z

__iter__ () → Iterator[float]

Returns iterable of x-, y- and z-axis.

__abs__ () → float

Returns length (magnitude) of vector.

replace (x: float = None, y: float = None, z: float = None) → Vec3

Returns a copy of vector with replaced x-, y- and/or z-axis.

classmethod generate (items: Iterable[Vertex]) → Iterable[Vec3]

Returns an iterable of `Vec3` objects.

classmethod list (items: Iterable[Vertex]) → List[Vec3]

Returns a list of `Vec3` objects.

classmethod tuple (items: Iterable[Vertex]) → Sequence[Vec3]

Returns a tuple of `Vec3` objects.

classmethod from_angle (angle: float, length: float = 1.) → Vec3

Returns a `Vec3` object from `angle` in radians in the xy-plane, z-axis = 0.

classmethod from_deg_angle (angle: float, length: float = 1.) → Vec3

Returns a `Vec3` object from `angle` in degrees in the xy-plane, z-axis = 0.

orthogonal (*ccw: bool = True*) → Vec3

Returns orthogonal 2D vector, z-axis is unchanged.

Parameters **ccw** – counter clockwise if True else clockwise

lerp (*other: Vertex, factor=.5*) → Vec3

Returns linear interpolation between *self* and *other*.

Parameters

- **other** – end point as *Vec3* compatible object

- **factor** – interpolation factor (0 = self, 1 = other, 0 . 5 = mid point)

is_parallel (*other: Vec3, abs_tol=1e-12*) → bool

Returns True if *self* and *other* are parallel to vectors.

project (*other: Vertex*) → Vec3

Returns projected vector of *other* onto *self*.

normalize (*length: float = 1.*) → Vec3

Returns normalized vector, optional scaled by *length*.

reversed () → Vec3

Returns negated vector (-*self*).

isclose (*other: Vertex, *, rel_tol: float = 1e-09, abs_tol: float = 1e-12*) → bool

Returns True if *self* is close to *other*. Uses math.isclose() to compare all axis.

Learn more about the math.isclose() function in [PEP 485](#).

__neg__ () → Vec3

Returns negated vector (-*self*).

__bool__ () → bool

Returns True if vector is not (0, 0, 0).

__eq__ (*other: Vertex*) → bool

Equal operator.

Parameters **other** – *Vec3* compatible object

__lt__ (*other: Vertex*) → bool

Lower than operator.

Parameters **other** – *Vec3* compatible object

__add__ (*other: Vertex*) → Vec3

Add *Vec3* operator: *self* + *other*.

__radd__ (*other: Vertex*) → Vec3

RAdd *Vec3* operator: *other* + *self*.

__sub__ (*other: Vertex*) → Vec3

Sub *Vec3* operator: *self* - *other*.

__rsub__ (*other: Vertex*) → Vec3

RSub *Vec3* operator: *other* - *self*.

__mul__ (*other: float*) → Vec3

Scalar Mul operator: *self* * *other*.

__rmul__ (*other: float*) → Vec3

Scalar RMul operator: *other* * *self*.

__truediv__ (*other: float*) → Vec3

Scalar Div operator: *self* / *other*.

dot (*other: Vertex*) → float

Dot operator: *self* . *other*

Parameters **other** – *Vec3* compatible object

cross (*other: Vertex*) → Vec3

Dot operator: *self* x *other*

Parameters **other** – *Vec3* compatible object

distance (*other: Vertex*) → float

Returns distance between *self* and *other* vector.

angle_about (*base: Vec3*, *target: Vec3*) → float

Returns counter clockwise angle in radians about *self* from *base* to *target* when projected onto the plane defined by *self* as the normal vector.

Parameters

- **base** – base vector, defines angle 0

- **target** – target vector

angle_between (*other: Vertex*) → float

Returns angle between *self* and *other* in radians. +angle is counter clockwise orientation.

Parameters **other** – *Vec3* compatible object

rotate (*angle: float*) → Vec3

Returns vector rotated about *angle* around the z-axis.

Parameters **angle** – angle in radians

rotate_deg (*angle: float*) → Vec3

Returns vector rotated about *angle* around the z-axis.

Parameters **angle** – angle in degrees

static sum (*items: Iterable[Vertex]*) → Vec3

Add all vectors in *items*.

ezdxf.math.X_AXIS

Vec3(1, 0, 0)

ezdxf.math.Y_AXIS

Vec3(0, 1, 0)

ezdxf.math.Z_AXIS

Vec3(0, 0, 1)

ezdxf.math.NULLVEC

Vec3(0, 0, 0)

Vec2

class eздxf.math.Vec2 (*v=(0.0, 0.0)*, *y=None*)

Vec2 represents a special 2D vector (x, y). The *Vec2* class is optimized for speed and not immutable, *iadd()*, *isub()*, *imul()* and *idiv()* modifies the vector itself, the *Vec3* class returns a new object.

`Vec2` initialization accepts float-tuples (`x, y[, z]`), two floats or any object providing `x` and `y` attributes like `Vec2` and `Vec3` objects.

Parameters

- `v` – vector object with `x` and `y` attributes/properties or a sequence of float [`x, y, ...`] or x-axis as float if argument `y` is not `None`
- `y` – second float for `Vec2(x, y)`

`Vec2` implements a subset of `Vec3`.

Plane

class `ezdxf.math.Plane(normal: Vec3, distance: float)`

Represents a plane in 3D space as normal vector and the perpendicular distance from origin.

`normal`

Normal vector of the plane.

`distance_from_origin`

The (perpendicular) distance of the plane from origin (0, 0, 0).

`vector`

Returns the location vector.

classmethod `from_3p(a: Vec3, b: Vec3, c: Vec3) → Plane`

Returns a new plane from 3 points in space.

classmethod `from_vector(vector) → Plane`

Returns a new plane from a location vector.

copy() → Plane

Returns a copy of the plane.

signed_distance_to(v: Vec3) → float

Returns signed distance of vertex `v` to plane, if distance is > 0, `v` is in ‘front’ of plane, in direction of the normal vector, if distance is < 0, `v` is at the ‘back’ of the plane, in the opposite direction of the normal vector.

distance_to(v: Vec3) → float

Returns absolute (unsigned) distance of vertex `v` to plane.

is_coplanar_vertex(v: Vec3, abs_tol=1e-9) → bool

Returns `True` if vertex `v` is coplanar, distance from plane to vertex `v` is 0.

is_coplanar_plane(p: Plane, abs_tol=1e-9) → bool

Returns `True` if plane `p` is coplanar, normal vectors in same or opposite direction.

BoundingBox

class `ezdxf.math.BoundingBox(vertices: Iterable[Vertex] = None)`

3D bounding box.

Parameters `vertices` – iterable of (`x, y, z`) tuples or `Vec3` objects

`extmin`

“lower left” corner of bounding box

extmax

“upper right” corner of bounding box

inside(*vertex*: Vertex) → bool

Returns True if *vertex* is inside this bounding box.

Vertices at the box border are inside!

any_inside(*vertices*: Iterable[Vertex]) → bool

Returns True if any vertex is inside this bounding box.

Vertices at the box border are inside!

all_inside(*vertices*: Iterable[Vertex]) → bool

Returns True if all vertices are inside this bounding box.

Vertices at the box border are inside!

intersect(*other*: BoundingBox) → bool

Returns True if this bounding box intersects with *other* but does not include touching bounding boxes,
see also [overlap\(\)](#):

```
bbox1 = BoundingBox([(0, 0, 0), (1, 1, 1)])
bbox2 = BoundingBox([(1, 1, 1), (2, 2, 2)])
assert bbox1.intersect(bbox2) is False
```

overlap(*other*: BoundingBox) → bool

Returns True if this bounding box intersects with *other* but in contrast to [intersect\(\)](#) includes touching bounding boxes too:

```
bbox1 = BoundingBox([(0, 0, 0), (1, 1, 1)])
bbox2 = BoundingBox([(1, 1, 1), (2, 2, 2)])
assert bbox1.overlap(bbox2) is True
```

New in version 0.17.2.

contains(*other*: BoundingBox) → bool

Returns True if the *other* bounding box is completely inside of this bounding box.

New in version 0.17.2.

extend(*vertices*: Iterable[Vertex]) → None

Extend bounds by *vertices*.

Parameters **vertices** – iterable of Vertex objects

union(*other*: BoundingBox) → BoundingBox

Returns a new bounding box as union of this and *other* bounding box.

rect_vertices() → Tuple[Vec2, ...]

Returns the corners of the bounding box in the xy-plane as [Vec2](#) objects.

cube_vertices() → Tuple[Vec3, ...]

Returns the 3D corners of the bounding box as [Vec3](#) objects.

BoundingBox2d

class ezdxf.math.BoundingBox2d(*vertices*: Iterable[Vertex] = None)

Optimized 2D bounding box.

Parameters **vertices** – iterable of (x, y[, z]) tuples or [Vec3](#) objects

extmin

“lower left” corner of bounding box

extmax

“upper right” corner of bounding box

inside(*vertex*: Vertex) → bool

Returns True if *vertex* is inside this bounding box.

Vertices at the box border are inside!

any_inside(*vertices*: Iterable[Vertex]) → bool

Returns True if any vertex is inside this bounding box.

Vertices at the box border are inside!

all_inside(*vertices*: Iterable[Vertex]) → bool

Returns True if all vertices are inside this bounding box.

Vertices at the box border are inside!

intersect(*other*: BoundingBox2d) → bool

Returns True if this bounding box intersects with *other* but does not include touching bounding boxes,
see also [overlap\(\)](#):

```
bbox1 = BoundingBox2d([(0, 0), (1, 1)])
bbox2 = BoundingBox2d([(1, 1), (2, 2)])
assert bbox1.intersect(bbox2) is False
```

overlap(*other*: BoundingBox2d) → bool

Returns True if this bounding box intersects with *other* but in contrast to [intersect\(\)](#) includes touching bounding boxes too:

```
bbox1 = BoundingBox2d([(0, 0), (1, 1)])
bbox2 = BoundingBox2d([(1, 1), (2, 2)])
assert bbox1.overlap(bbox2) is True
```

New in version 0.17.2.

contains(*other*: BoundingBox2d) → bool

Returns True if the *other* bounding box is completely inside of this bounding box.

New in version 0.17.2.

extend(*vertices*: Iterable[Vertex]) → None

Extend bounds by *vertices*.

Parameters **vertices** – iterable of Vertex objects

union(*other*: BoundingBox2d) → BoundingBox2d

Returns a new bounding box as union of this and *other* bounding box.

rect_vertices() → Tuple[Vec2, ...]

Returns the corners of the bounding box in the xy-plane as [Vec2](#) objects.

ConstructionRay

class ezdxf.math.**ConstructionRay**(*p1*: Vertex, *p2*: Vertex = None, *angle*: float = None)

Infinite 2D construction ray as immutable object.

Parameters

- **p1** – definition point 1
- **p2** – ray direction as 2nd point or None
- **angle** – ray direction as angle in radians or None

location

Location vector as `Vec2`.

direction

Direction vector as `Vec2`.

slope

Slope of ray or None if vertical.

angle

Angle between x-axis and ray in radians.

angle_deg

Angle between x-axis and ray in degrees.

is_vertical

True if ray is vertical (parallel to y-axis).

is_horizontal

True if ray is horizontal (parallel to x-axis).

__str__()

Return str(self).

is_parallel (self, other: ConstructionRay) → bool

Returns True if rays are parallel.

intersect (other: ConstructionRay) → Vec2

Returns the intersection point as (x, y) tuple of *self* and *other*.

Raises ParallelRaysError – if rays are parallel

orthogonal (location: 'Vertex') → ConstructionRay

Returns orthogonal ray at *location*.

bisectrix (other: ConstructionRay) → ConstructionRay:

Bisectrix between *self* and *other*.

yof (x: float) → float

Returns y-value of ray for *x* location.

Raises ArithmeticError – for vertical rays

xof (y: float) → float

Returns x-value of ray for *y* location.

Raises ArithmeticError – for horizontal rays

ConstructionLine

class ezdxf.math.ConstructionLine (start: Vertex, end: Vertex)

2D ConstructionLine is similar to `ConstructionRay`, but has a start- and endpoint. The direction of line goes from start- to endpoint, “left of line” is always in relation to this line direction.

Parameters

- **start** – start point of line as `Vec2` compatible object

- **end** – end point of line as `Vec2` compatible object

start

start point as `Vec2`

end

end point as `Vec2`

bounding_box

bounding box of line as `BoundingBox2d` object.

ray

collinear `ConstructionRay`.

is_vertical

True if line is vertical.

is_horizontal

True if line is horizontal.

__str__()

Return str(self).

translate (*dx: float, dy: float*) → None

Move line about *dx* in x-axis and about *dy* in y-axis.

Parameters

- **dx** – translation in x-axis
- **dy** – translation in y-axis

length() → float

Returns length of line.

midpoint() → Vec2

Returns mid point of line.

inside_bounding_box (*point: Vertex*) → bool

Returns True if *point* is inside of line bounding box.

intersect (*other: ConstructionLine, abs_tol:float=1e-10*) → Optional[`Vec2`]

Returns the intersection point of to lines or None if they have no intersection point.

Parameters

- **other** – other `ConstructionLine`
- **abs_tol** – tolerance for distance check

has_intersection (*other: ConstructionLine, abs_tol:float=1e-10*) → bool

Returns True if has intersection with *other* line.

is_point_left_of_line (*point: Vertex, colinear=False*) → bool

Returns True if *point* is left of construction line in relation to the line direction from start to end.

If *colinear* is True, a colinear point is also left of the line.

ConstructionCircle

class `ezdxf.math.ConstructionCircle` (*center: Vertex, radius: float = 1.0*)

Circle construction tool.

Parameters

- **center** – center point as `Vec2` compatible object
- **radius** – circle radius > 0

center

center point as `Vec2`

radius

radius as float

bounding_box

2D bounding box of circle as `BoundingBox2d` object.

static from_3p (*p1: Vertex, p2: Vertex, p3: Vertex*) → `ConstructionCircle`

Creates a circle from three points, all points have to be compatible to `Vec2` class.

__str__ () → str

Returns string representation of circle “`ConstructionCircle(center, radius)`”.

translate (*dx: float, dy: float*) → None

Move circle about *dx* in x-axis and about *dy* in y-axis.

Parameters

- **dx** – translation in x-axis
- **dy** – translation in y-axis

point_at (*angle: float*) → `Vec2`

Returns point on circle at *angle* as `Vec2` object.

Parameters angle – angle in radians, angle goes counter clockwise around the z-axis, x-axis = 0 deg.

vertices (*angles: Iterable[float]*) → `Iterator[Vec2]`

Yields vertices of the circle for iterable *angles*.

Parameters angles – iterable of angles as radians, angle goes counter clockwise around the z-axis, x-axis = 0 deg.

New in version 0.17.1.

flattening (*sagitta: float*) → `Iterator[Vec2]`

Approximate the circle by vertices, argument *sagitta* is the max. distance from the center of an arc segment to the center of its chord. Returns a closed polygon where the start vertex is coincident with the end vertex!

New in version 0.17.1.

inside (*point: Vertex*) → bool

Returns True if *point* is inside circle.

tangent (*angle: float*) → `ConstructionRay`

Returns tangent to circle at *angle* as `ConstructionRay` object.

Parameters angle – angle in radians

intersect_ray (*ray: ConstructionRay, abs_tol: float = 1e-10*) → `Sequence[Vec2]`

Returns intersection points of circle and *ray* as sequence of `Vec2` objects.

Parameters

- **ray** – intersection ray
- **abs_tol** – absolute tolerance for tests (e.g. test for tangents)

Returnstuple of `Vec2` objects

tuple size	Description
0	no intersection
1	ray is a tangent to circle
2	ray intersects with the circle

intersect_line (*ray: ConstructionLine, abs_tol: float = 1e-10*) → Sequence[`Vec2`]Returns intersection points of circle and *line* as sequence of `Vec2` objects.**Parameters**

- **line** – intersection line
- **abs_tol** – absolute tolerance for tests (e.g. test for tangents)

Returnstuple of `Vec2` objects

tuple size	Description
0	no intersection
1	line intersects or touches the circle at one point
2	line intersects the circle at two points

New in version 0.17.1.

intersect_circle (*other: ConstructionCircle, abs_tol: float = 1e-10*) → Sequence[`Vec2`]Returns intersection points of two circles as sequence of `Vec2` objects.**Parameters**

- **other** – intersection circle
- **abs_tol** – absolute tolerance for tests

Returnstuple of `Vec2` objects

tuple size	Description
0	no intersection
1	circle touches the <i>other</i> circle at one point
2	circle intersects with the <i>other</i> circle

ConstructionArc

class `ezdxf.math.ConstructionArc` (*center: Vertex = (0, 0), radius: float = 1.0, start_angle: float = 0.0, end_angle: float = 360.0, is_counter_clockwise: bool = True*)This is a helper class to create parameters for the DXF `Arc` class.*ConstructionArc* represents a 2D arc in the xy-plane, use an `UCS` to place arc in 3D space, see method `add_to_layout()`.Implements the 2D transformation tools: `translate()`, `scale_uniform()` and `rotate_z()`

Parameters

- **center** – center point as `Vec2` compatible object
- **radius** – radius
- **start_angle** – start angle in degrees
- **end_angle** – end angle in degrees
- **is_counter_clockwise** – swaps start- and end angle if `False`

center

center point as `Vec2`

radius

radius as float

start_angle

start angle in degrees

end_angle

end angle in degrees

angle_span

Returns angle span of arc from start- to end param.

start_angle_rad

Returns the start angle in radians.

end_angle_rad

Returns the end angle in radians.

start_point

start point of arc as `Vec2`.

end_point

end point of arc as `Vec2`.

bounding_box

bounding box of arc as `BoundingBox2d`.

angles (*num: int*) → Iterable[float]

Returns *num* angles from start- to end angle in degrees in counter clockwise order.

All angles are normalized in the range from [0, 360).

vertices (*a: Iterable[float]*) → Iterable[ezdxf.math._vector.Vec2]

Yields vertices on arc for angles in iterable *a* in WCS as location vectors.

Parameters *a* – angles in the range from 0 to 360 in degrees, arc goes counter clockwise around the z-axis, WCS x-axis = 0 deg.

tangents (*a: Iterable[float]*) → Iterable[ezdxf.math._vector.Vec2]

Yields tangents on arc for angles in iterable *a* in WCS as direction vectors.

Parameters *a* – angles in the range from 0 to 360 in degrees, arc goes counter clockwise around the z-axis, WCS x-axis = 0 deg.

translate (*dx: float, dy: float*) → ConstructionArc

Move arc about *dx* in x-axis and about *dy* in y-axis, returns *self* (floating interface).

Parameters

- **dx** – translation in x-axis

- **dy** – translation in y-axis

scale_uniform(*s*: float) → ConstructionArc

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

rotate_z(*angle*: float) → ConstructionArc

Rotate arc inplace about z-axis, returns *self* (floating interface).

Parameters **angle** – rotation angle in degrees

classmethod from_2p_angle(*start_point*: Vertex, *end_point*: Vertex, *angle*: float, *ccw*: bool = True) → ConstructionArc

Create arc from two points and enclosing angle. Additional precondition: arc goes by default in counter clockwise orientation from *start_point* to *end_point*, can be changed by *ccw* = False.

Parameters

- **start_point** – start point as *Vec2* compatible object
- **end_point** – end point as *Vec2* compatible object
- **angle** – enclosing angle in degrees
- **ccw** – counter clockwise direction if True

classmethod from_2p_radius(*start_point*: Vertex, *end_point*: Vertex, *radius*: float, *ccw*: bool = True, *center_is_left*: bool = True) → ConstructionArc

Create arc from two points and arc radius. Additional precondition: arc goes by default in counter clockwise orientation from *start_point* to *end_point* can be changed by *ccw* = False.

The parameter *center_is_left* defines if the center of the arc is left or right of the line from *start_point* to *end_point*. Parameter *ccw* = False swaps start- and end point, which also inverts the meaning of *center_is_left*.

Parameters

- **start_point** – start point as *Vec2* compatible object
- **end_point** – end point as *Vec2* compatible object
- **radius** – arc radius
- **ccw** – counter clockwise direction if True
- **center_is_left** – center point of arc is left of line from start- to end point if True

classmethod from_3p(*start_point*: Vertex, *end_point*: Vertex, *def_point*: Vertex, *ccw*: bool = True) → ConstructionArc

Create arc from three points. Additional precondition: arc goes in counter clockwise orientation from *start_point* to *end_point*.

Parameters

- **start_point** – start point as *Vec2* compatible object
- **end_point** – end point as *Vec2* compatible object
- **def_point** – additional definition point as *Vec2* compatible object
- **ccw** – counter clockwise direction if True

add_to_layout(*layout*: BaseLayout, *ucs*: UCS = None, *dxftattribs*: dict = None) → Arc

Add arc as DXF *Arc* entity to a layout.

Supports 3D arcs by using an *UCS*. An *ConstructionArc* is always defined in the xy-plane, but by using an arbitrary UCS, the arc can be placed in 3D space, automatically OCS transformation included.

Parameters

- **layout** – destination layout as `BaseLayout` object
- **ucs** – place arc in 3D space by `UCS` object
- **dxfattribs** – additional DXF attributes for the ARC entity

intersect_ray (*ray*: `ConstructionRay`, *abs_tol*: `float = 1e-10`) → `Sequence[Vec2]`

Returns intersection points of arc and *ray* as sequence of `Vec2` objects.

Parameters

- **ray** – intersection ray
- **abs_tol** – absolute tolerance for tests (e.g. test for tangents)

Returns

tuple of `Vec2` objects

tuple size	Description
0	no intersection
1	line intersects or touches the arc at one point
2	line intersects the arc at two points

New in version 0.17.1.

intersect_line (*ray*: `ConstructionLine`, *abs_tol*: `float = 1e-10`) → `Sequence[Vec2]`

Returns intersection points of arc and *line* as sequence of `Vec2` objects.

Parameters

- **line** – intersection line
- **abs_tol** – absolute tolerance for tests (e.g. test for tangents)

Returns

tuple of `Vec2` objects

tuple size	Description
0	no intersection
1	line intersects or touches the arc at one point
2	line intersects the arc at two points

New in version 0.17.1.

intersect_circle (*circle*: `ConstructionCircle`, *abs_tol*: `float = 1e-10`) → `Sequence[Vec2]`

Returns intersection points of arc and *circle* as sequence of `Vec2` objects.

Parameters

- **circle** – intersection circle
- **abs_tol** – absolute tolerance for tests

Returns

tuple of `Vec2` objects

tuple size	Description
0	no intersection
1	circle intersects or touches the arc at one point
2	circle intersects the arc at two points

New in version 0.17.1.

intersect_arc (*other*: *ConstructionArc*, *abs_tol*: *float* = *1e-10*) → *Sequence[Vec2]*
Returns intersection points of two arcs as sequence of *Vec2* objects.

Parameters

- **other** – other intersection arc
- **abs_tol** – absolute tolerance for tests

Returns

tuple of *Vec2* objects

tuple size	Description
0	no intersection
1	other arc intersects or touches the arc at one point
2	other arc intersects the arc at two points

New in version 0.17.1.

ConstructionEllipse

```
class ezdxf.math.ConstructionEllipse(center: Vertex = Vec3(0.0, 0.0, 0.0), major_axis: Vertex = Vec3(1.0, 0.0, 0.0), extrusion: Vertex = Vec3(0.0, 0.0, 1.0), ratio: float = 1, start_param: float = 0, end_param: float = 6.283185307179586, ccw: bool = True)
```

This is a helper class to create parameters for 3D ellipses.

Parameters

- **center** – 3D center point
- **major_axis** – major axis as 3D vector
- **extrusion** – normal vector of ellipse plane
- **ratio** – ratio of minor axis to major axis
- **start_param** – start param in radians
- **end_param** – end param in radians
- **ccw** – is counter clockwise flag - swaps start- and end param if `False`

center

center point as *Vec3*

major_axis

major axis as *Vec3*

minor_axis

minor axis as *Vec3*, automatically calculated from *major_axis* and *extrusion*.

extrusion

extrusion vector (normal of ellipse plane) as *Vec3*

ratio

ratio of minor axis to major axis (float)

```

start
    start param in radians (float)

end
    end param in radians (float)

start_point
    Returns start point of ellipse as Vec3.

end_point
    Returns end point of ellipse as Vec3.

to_ocs () → ConstructionEllipse
    Returns ellipse parameters as OCS representation.
    OCS elevation is stored in center.z.

params (num: int) → Iterable[float]
    Returns num params from start- to end param in counter clockwise order.
    All params are normalized in the range from [0, 2π).

vertices (params: Iterable[float]) → Iterable[ezdxf.math._vector.Vec3]
    Yields vertices on ellipse for iterable params in WCS.

    Parameters params – param values in the range from [0, 2π) in radians, param goes counter
    clockwise around the extrusion vector, major_axis = local x-axis = 0 rad.

flattening (distance: float, segments: int = 4) → Iterable[ezdxf.math._vector.Vec3]
    Adaptive recursive flattening. The argument segments is the minimum count of approximation segments, if
    the distance from the center of the approximation segment to the curve is bigger than distance the segment
    will be subdivided. Returns a closed polygon for a full ellipse: start vertex == end vertex.

    Parameters
        • distance – maximum distance from the projected curve point onto the segment chord.
        • segments – minimum segment count

    New in version 0.15.

params_from_vertices (vertices: Iterable[Vertex]) → Iterable[float]
    Yields ellipse params for all given vertices.
    The vertex don't has to be exact on the ellipse curve or in the range from start- to end param or even in the
    ellipse plane. Param is calculated from the intersection point of the ray projected on the ellipse plane from
    the center of the ellipse through the vertex.



Warning: An input for start- and end vertex at param 0 and 2π return unpredictable results because
of floating point inaccuracy, sometimes 0 and sometimes 2π.

dxfattribs () → Dict[KT, VT]
    Returns required DXF attributes to build an ELLIPSE entity.
    Entity ELLIPSE has always a ratio in range from 1e-6 to 1.

main_axis_points () → Iterable[ezdxf.math._vector.Vec3]
    Yields main axis points of ellipse in the range from start- to end param.

```

```
classmethod from_arc(center: Vertex=(0, 0, 0), radius: float = 1, extrusion: Vertex=(0, 0, 1),
                     start_angle: float = 0, end_angle: float = 360, ccw: bool = True) →
    ConstructionEllipse
```

Returns `ConstructionEllipse` from arc or circle.

Arc and Circle parameters defined in OCS.

Parameters

- **center** – center in OCS
- **radius** – arc or circle radius
- **extrusion** – OCS extrusion vector
- **start_angle** – start angle in degrees
- **end_angle** – end angle in degrees
- **ccw** – arc curve goes counter clockwise from start to end if True

```
transform(m: Matrix44)
```

Transform ellipse in place by transformation matrix *m*.

```
swap_axis() → None
```

Swap axis and adjust start- and end parameter.

```
add_to_layout(layout: BaseLayout, dxfattribs: dict = None) → Ellipse
```

Add ellipse as DXF `Ellipse` entity to a layout.

Parameters

- **layout** – destination layout as `BaseLayout` object
- **dxfattribs** – additional DXF attributes for the ELLIPSE entity

ConstructionBox

```
class ezdxf.math.ConstructionBox(center: Vertex = (0, 0), width: float = 1, height: float = 1,
                                   angle: float = 0)
```

Helper class to create rectangles.

Parameters

- **center** – center of rectangle
- **width** – width of rectangle
- **height** – height of rectangle
- **angle** – angle of rectangle in degrees

center

box center

width

box width

height

box height

angle

rotation angle in degrees

corners

box corners as sequence of `Vec2` objects.

bounding_box

`BoundingBox2d`

incircle_radius

incircle radius

circumcircle_radius

circum circle radius

__iter__ () → Iterable[Vec2]

Iterable of box corners as `Vec2` objects.

__getitem__ (corner) → Vec2

Get corner by index `corner`, list like slicing is supported.

__repr__ () → str

Returns string representation of box as `ConstructionBox(center, width, height, angle)`

classmethod from_points (p1: Vertex, p2: Vertex) → ConstructionBox

Creates a box from two opposite corners, box sides are parallel to x- and y-axis.

Parameters

- `p1` – first corner as `Vec2` compatible object
- `p2` – second corner as `Vec2` compatible object

translate (dx: float, dy: float) → None

Move box about `dx` in x-axis and about `dy` in y-axis.

Parameters

- `dx` – translation in x-axis
- `dy` – translation in y-axis

expand (dw: float, dh: float) → None

Expand box: `dw` expand width, `dh` expand height.

scale (sw: float, sh: float) → None

Scale box: `sw` scales width, `sh` scales height.

rotate (angle: float) → None

Rotate box by `angle` in degrees.

is_inside (point: Vertex) → bool

Returns True if `point` is inside of box.

is_any_corner_inside (other: ConstructionBox) → bool

Returns True if any corner of `other` box is inside this box.

is_overlapping (other: ConstructionBox) → bool

Returns True if this box and `other` box do overlap.

border_lines () → Sequence[ConstructionLine]

Returns border lines of box as sequence of `ConstructionLine`.

intersect (line: ConstructionLine) → List[Vec2]

Returns 0, 1 or 2 intersection points between `line` and box border lines.

Parameters `line` – line to intersect with border lines

Returns

list of intersection points

list size	Description
0	no intersection
1	line touches box at one corner
2	line intersects with box

ConstructionPolyline

```
class ezdxf.math.ConstructionPolyline(vertices: Iterable[Union[Sequence[float],  
ezdxf.math._vector.Vec2, ezdxf.math._vector.Vec3]],  
close: bool = False, rel_tol: float = 1e-09)
```

A polyline construction tool to measure, interpolate and divide anything that can be approximated or flattened into vertices. This is an immutable data structure which supports the Sequence interface.

Parameters

- **vertices** – iterable of polyline vertices
- **close** – True to close the polyline (first vertex == last vertex)
- **rel_tol** – relative tolerance for floating point comparisons

Example to measure or divide a SPLINE entity:

```
import ezdxf  
from ezdxf.math import ConstructionPolyline  
  
doc = ezdxf.readfile("your.dxf")  
msp = doc.modelspace()  
spline = msp.query("SPLINE").first  
if spline is not None:  
    polyline = ConstructionPolyline(spline.flattening(0.01))  
    print(f"Entity {spline} has an approximated length of {polyline.length}")  
    # get dividing points with a distance of 1.0 drawing unit to each other  
    points = list(polyline.divide_by_length(1.0))
```

New in version 0.18.

data (index: int) → Tuple[float, float, Vec3]

Returns the tuple (distance from start, distance from previous vertex, vertex). All distances measured along the polyline.

index_at (distance: float) → int

Returns the data index of the exact or next data entry for the given *distance*. Returns the index of last entry if *distance* > length.

vertex_at (distance: float) → Vec3

Returns the interpolated vertex at the given *distance* from the start of the polyline.

divide (count: int) → Iterator[Vec3]

Returns *count* interpolated vertices along the polyline. Argument *count* has to be greater than 2 and the start- and end vertices are always included.

divide_by_length (length: float, force_last: bool = False) → Iterator[Vec3]

Returns interpolated vertices along the polyline. Each vertex has a fix distance *length* from its predecessor. Yields the last vertex if argument *force_last* is True even if the last distance is not equal to *length*.

Shape2d

```
class ezdxf.math.Shape2d(vertices: Iterable[Vertex] = None)
    2D geometry object as list of Vec2 objects, vertices can be moved, rotated and scaled.

    Parameters vertices – iterable of Vec2 compatible objects.

    vertices
        List of Vec2 objects

    bounding_box
        BoundingBox2d

    __len__ () → int
        Returns count of vertices.

    __getitem__ (item) → Vec2
        Get vertex by index item, supports list like slicing.

    append (vertex: Vertex) → None
        Append single vertex.

        Parameters vertex – vertex as Vec2 compatible object

    extend (vertices: Iterable[T_co]) → None
        Append multiple vertices.

        Parameters vertices – iterable of vertices as Vec2 compatible objects

    translate (vector: Vertex) → None
        Translate shape about vector.

    scale (sx: float = 1.0, sy: float = 1.0) → None
        Scale shape about sx in x-axis and sy in y-axis.

    scale_uniform (scale: float) → None
        Scale shape uniform about scale in x- and y-axis.

    rotate (angle: float, center: Vertex = None) → None
        Rotate shape around rotation center about angle in degrees.

    rotate_rad (angle: float, center: Vertex = None) → None
        Rotate shape around rotation center about angle in radians.

    offset (offset: float, closed: bool = False) → ezdxf.math.shape.Shape2d
        Returns a new offset shape, for more information see also ezdxf.math.offset\_vertices\_2d\(\) function.

        Parameters
            • offset – line offset perpendicular to direction of shape segments defined by vertices order, offset > 0 is ‘left’ of line segment, offset < 0 is ‘right’ of line segment
            • closed – True to handle as closed shape

    convex_hull () → ezdxf.math.shape.Shape2d
        Returns convex hull as new shape.
```

Curves

BSpline

```
class ezdxf.math.BSpline(control_points: Iterable[Vertex], order: int = 4, knots: Iterable[float] = None, weights: Iterable[float] = None)
```

Representation of a B-spline curve. The default configuration of the knot vector is an uniform open `knot` vector (“clamped”).

Factory functions:

- `fit_points_to_cad_cv()`
- `fit_points_to_cubic_bezier()`
- `open_uniform_bspline()`
- `closed_uniform_bspline()`
- `rational_bspline_from_arc()`
- `rational_bspline_from_ellipse()`
- `global_bspline_interpolation()`
- `local_cubic_bspline_interpolation()`

Parameters

- `control_points` – iterable of control points as `Vec3` compatible objects
- `order` – spline order (degree + 1)
- `knots` – iterable of knot values
- `weights` – iterable of weight values

`knots()` → Tuple[float, ...]

Returns a tuple of `knot` values as floats, the knot vector **always** has order + count values ($n + p + 2$ in text book notation).

`weights()` → Tuple[float, ...]

Returns a tuple of weights values as floats, one for each control point or an empty tuple.

`params(segments: int)` → Iterable[float]

Yield evenly spaced parameters for given segment count.

`reverse()` → BSpline

Returns a new `BSpline` object with reversed control point order.

`transform(m: Matrix44)` → BSpline

Returns a new `BSpline` object transformed by a `Matrix44` transformation matrix.

`approximate(segments: int = 20)` → Iterable[`Vec3`]

Approximates curve by vertices as `Vec3` objects, vertices count = segments + 1.

`flattening(distance: float, segments: int = 4)` → Iterable[`Vec3`]

Adaptive recursive flattening. The argument `segments` is the minimum count of approximation segments between two knots, if the distance from the center of the approximation segment to the curve is bigger than `distance` the segment will be subdivided.

Parameters

- `distance` – maximum distance from the projected curve point onto the segment chord.
- `segments` – minimum segment count between two knots

New in version 0.15.

point (*t*: float) → Vec3

Returns point for parameter *t*.

Parameters **t** – parameter in range [0, max_t]

points (*t*: Iterable[float]) → List[Vec3]

Yields points for parameter vector *t*.

Parameters **t** – parameters in range [0, max_t]

derivative (*t*: float, *n*: int=2) → List[Vec3]

Return point and derivatives up to *n* <= degree for parameter *t*.

e.g. *n*=1 returns point and 1st derivative.

Parameters

- **t** – parameter in range [0, max_t]
- **n** – compute all derivatives up to n <= degree

Returns *n*+1 values as *Vec3* objects

derivatives (*t*: Iterable[float], *n*: int=2) → Iterable[List[Vec3]]

Yields points and derivatives up to *n* <= degree for parameter vector *t*.

e.g. *n*=1 returns point and 1st derivative.

Parameters

- **t** – parameters in range [0, max_t]
- **n** – compute all derivatives up to n <= degree

Returns List of *n*+1 values as *Vec3* objects

insert_knot (*t*: float) → BSpline

Insert an additional knot, without altering the shape of the curve. Returns a new *BSpline* object.

Parameters **t** – position of new knot $0 < t < \text{max_t}$

knot_refinement (*u*: Iterable[float]) → BSpline

Insert multiple knots, without altering the shape of the curve. Returns a new *BSpline* object.

Parameters **u** – vector of new knots *t* and for each *t*: $0 < t < \text{max_t}$

static from_ellipse (*ellipse*: ConstructionEllipse) → BSpline

Returns the ellipse as *BSpline* of 2nd degree with as few control points as possible.

static from_arc (*arc*: ConstructionArc) → BSpline

Returns the arc as *BSpline* of 2nd degree with as few control points as possible.

static from_fit_points (*points*: Iterable[Vertex], *degree*: int=3, *method*=’chord’) → BSpline

Returns *BSpline* defined by fit points.

static arc_approximation (*arc*: ConstructionArc, *num*: int=16) → BSpline

Returns an arc approximation as *BSpline* with *num* control points.

static ellipse_approximation (*ellipse*: ConstructionEllipse, *num*: int=16) → BSpline

Returns an ellipse approximation as *BSpline* with *num* control points.

bezier_decomposition () → Iterable[List[Vec3]]

Decompose a non-rational B-spline into multiple Bézier curves.

This is the preferred method to represent the most common non-rational B-splines of 3rd degree by cubic Bézier curves, which are often supported by render backends.

Returns Yields control points of Bézier curves, each Bézier segment has degree+1 control points
e.g. B-spline of 3rd degree yields cubic Bézier curves of 4 control points.

cubic_bezier_approximation (*level: int = 3, segments: int = None*) → Iterable[Bezier4P]

Approximate arbitrary B-splines (degree != 3 and/or rational) by multiple segments of cubic Bézier curves. The choice of cubic Bézier curves is based on the widely support of this curves by many render backends. For cubic non-rational B-splines, which is maybe the most common used B-spline, is [bezier_decomposition\(\)](#) the better choice.

1. approximation by *level*: an educated guess, the first level of approximation segments is based on the count of control points and their distribution along the B-spline, every additional level is a subdivision of the previous level.

E.g. a B-Spline of 8 control points has 7 segments at the first level, 14 at the 2nd level and 28 at the 3rd level, a level >= 3 is recommended.

2. approximation by a given count of evenly distributed approximation segments.

Parameters

- **level** – subdivision level of approximation segments (ignored if argument *segments* is not *None*)
- **segments** – absolute count of approximation segments

Returns Yields control points of cubic Bézier curves as *Bezier4P* objects

Bezier

class `ezdxf.math.Bezier(defpoints: Iterable[Vertex])`

A Bézier curve is a parametric curve used in computer graphics and related fields. Bézier curves are used to model smooth curves that can be scaled indefinitely. “Paths”, as they are commonly referred to in image manipulation programs, are combinations of linked Bézier curves. Paths are not bound by the limits of rasterized images and are intuitive to modify. (Source: Wikipedia)

This is a generic implementation which works with any count of definition points greater than 2, but it is a simple and slow implementation. For more performance look at the specialized *Bezier4P* class.

Objects are immutable.

Parameters **defpoints** – iterable of definition points as *Vec3* compatible objects.

control_points

Control points as tuple of *Vec3* objects.

params (*segments: int*) → Iterable[float]

Yield evenly spaced parameters from 0 to 1 for given segment count.

reverse () → Bezier

Returns a new Bézier-curve with reversed control point order.

transform (*m: Matrix44*) → Bezier

General transformation interface, returns a new *Bezier* curve.

Parameters **m** – 4x4 transformation matrix (*ezdxf.math.Matrix44*)

approximate (*segments: int = 20*) → Iterable[*Vec3*]

Approximates curve by vertices as *Vec3* objects, vertices count = segments + 1.

flattening (*distance: float, segments: int=4*) → Iterable[Vec3]

Adaptive recursive flattening. The argument *segments* is the minimum count of approximation segments, if the distance from the center of the approximation segment to the curve is bigger than *distance* the segment will be subdivided.

Parameters

- **distance** – maximum distance from the center of the curve (Cn) to the center of the linear (C1) curve between two approximation points to determine if a segment should be subdivided.
- **segments** – minimum segment count

New in version 0.15.

point (*t: float*) → Vec3

Returns a point for parameter *t* in range [0, 1] as *Vec3* object.

points (*t: Iterable[float]*) → Iterable[Vec3]

Yields multiple points for parameters in vector *t* as *Vec3* objects. Parameters have to be in range [0, 1].

derivative (*t: float*) → Tuple[Vec3, Vec3, Vec3]

Returns (point, 1st derivative, 2nd derivative) tuple for parameter *t* in range [0, 1] as *Vec3* objects.

derivatives (*t: Iterable[float]*) → Iterable[Tuple[Vec3, Vec3, Vec3]]

Returns multiple (point, 1st derivative, 2nd derivative) tuples for parameter vector *t* as *Vec3* objects.
Parameters in range [0, 1]

Bezier4P

class ezdxf.math.Bezier4P (*defpoints: Sequence[Vertex]*)

Implements an optimized cubic Bézier curve for exact 4 control points.

A Bézier curve is a parametric curve, parameter *t* goes from 0 to 1, where 0 is the first control point and 1 is the fourth control point.

Special behavior:

- 2D control points in, returns 2D results as *Vec2* objects
- 3D control points in, returns 3D results as *Vec3* objects
- Object is immutable.

Parameters **defpoints** – iterable of definition points as *Vec2* or *Vec3* compatible objects.

control_points

Control points as tuple of *Vec3* or *Vec2* objects.

reverse () → Bezier4P

Returns a new Bézier-curve with reversed control point order.

transform (*m: Matrix44*) → Bezier4P

General transformation interface, returns a new Bezier4P curve and it is always a 3D curve.

Parameters **m** – 4x4 transformation matrix (*ezdxf.math.Matrix44*)

New in version 0.14.

approximate (*segments: int*) → Iterable[Union[Vec3, Vec2]]

Approximate Bézier curve by vertices, yields *segments* + 1 vertices as (x, y[, z]) tuples.

Parameters **segments** – count of segments for approximation

flattening (*distance: float, segments: int=4*) → Iterable[Union[Vec3, Vec2]]

Adaptive recursive flattening. The argument *segments* is the minimum count of approximation segments, if the distance from the center of the approximation segment to the curve is bigger than *distance* the segment will be subdivided.

Parameters

- **distance** – maximum distance from the center of the cubic (C3) curve to the center of the linear (C1) curve between two approximation points to determine if a segment should be subdivided.
- **segments** – minimum segment count

New in version 0.15.

approximated_length (*segments: int = 128*) → float

Returns estimated length of Bézier-curve as approximation by line *segments*.

point (*t: float*) → Union[Vec3, Vec2]

Returns point for location *t* at the Bézier-curve.

Parameters **t** – curve position in the range [0, 1]

tangent (*t: float*) → Union[Vec3, Vec2]

Returns direction vector of tangent for location *t* at the Bézier-curve.

Parameters **t** – curve position in the range [0, 1]

Bezier3P

class eздxf.math.Bezier3P (*defpoints: Sequence[Vertex]*)

Implements an optimized quadratic Bézier curve for exact 3 control points.

Special behavior:

- 2D control points in, returns 2D results as *Vec2* objects
- 3D control points in, returns 3D results as *Vec3* objects
- Object is immutable.

Parameters **defpoints** – iterable of definition points as *Vec2* or *Vec3* compatible objects.

control_points

Control points as tuple of *Vec3* or *Vec2* objects.

reverse () → Bezier3P

Returns a new Bézier-curve with reversed control point order.

transform (*m: Matrix44*) → Bezier3P

General transformation interface, returns a new *Bezier3P* curve and it is always a 3D curve.

Parameters **m** – 4x4 transformation matrix (*ezdxf.math.Matrix44*)

approximate (*segments: int*) → Iterable[Union[Vec3, Vec2]]

Approximate Bézier curve by vertices, yields *segments* + 1 vertices as (x, y[, z]) tuples.

Parameters **segments** – count of segments for approximation

flattening (*distance*: float, *segments*: int=4) → Iterable[Union[Vec3, Vec2]]

Adaptive recursive flattening. The argument *segments* is the minimum count of approximation segments, if the distance from the center of the approximation segment to the curve is bigger than *distance* the segment will be subdivided.

Parameters

- **distance** – maximum distance from the center of the quadratic (C2) curve to the center of the linear (C1) curve between two approximation points to determine if a segment should be subdivided.
- **segments** – minimum segment count

approximated_length (*segments*: int = 128) → float

Returns estimated length of Bézier-curve as approximation by line *segments*.

point (*t*: float) → Union[Vec3, Vec2]

Returns point for location *t* at the Bézier-curve.

Parameters **t** – curve position in the range [0, 1]

tangent (*t*: float) → Union[Vec3, Vec2]

Returns direction vector of tangent for location *t* at the Bézier-curve.

Parameters **t** – curve position in the range [0, 1]

ApproxParamT

class eздxf.math.**ApproxParamT** (*curve*, *, *max_t*: float = 1.0, *segments*: int = 100)

Approximation tool for parametrized curves.

- approximate parameter *t* for a given distance from the start of the curve
- approximate the distance for a given parameter *t* from the start of the curve

This approximations can be applied to all parametrized curves which provide a `point()` method, like `Bezier4P`, `Bezier3P` and `BSpline`.

The approximation is based on equally spaced parameters from 0 to *max_t* for a given segment count. The `flattening()` method can not be used for the curve approximation, because the required parameter *t* is not logged by the flattening process.

Parameters

- **curve** – curve object, requires a method `point()`
- **max_t** – the max. parameter value
- **segments** – count of approximation segments

New in version 0.18.

param_t (*distance*: float)

Approximate parameter *t* for the given *distance* from the start of the curve.

distance (*t*: float) → float

Approximate the distance from the start of the curve to the point *t* on the curve.

BezierSurface

```
class ezdxf.math.BezierSurface (defpoints: List[List[Vertex]])
```

BezierSurface defines a mesh of $m \times n$ control points. This is a parametric surface, which means the m -dimension goes from 0 to 1 as parameter u and the n -dimension goes from 0 to 1 as parameter v .

Parameters **defpoints** – matrix (list of lists) of m rows and n columns: [[m1n1, m1n2, ...],
[m2n1, m2n2, ...] ...] each element is a 3D location as (x , y , z) tuple.

nrows

count of rows (m-dimension)

ncols

count of columns (n-dimension)

point ($u: float, v: float$) → ezdxf.math._vector.Vec3

Returns a point for location (u, v) at the Bézier surface as (x , y , z) tuple, parameters u and v in the range of [0, 1].

approximate ($usegs: int, vsegs: int$) → List[List[ezdxf.math._vector.Vec3]]

Approximate surface as grid of (x , y , z) *Vec3*.

Parameters

- **usegs** – count of segments in u -direction (m-dimension)
- **vsegs** – count of segments in v -direction (n-dimension)

Returns list of $usegs + 1$ rows, each row is a list of $vsegs + 1$ vertices as *Vec3*.

EulerSpiral

```
class ezdxf.math.EulerSpiral (curvature: float = 1.0)
```

This class represents an euler spiral (clothoid) for *curvature* (Radius of curvature).

This is a parametric curve, which always starts at the origin = (0, 0).

Parameters **curvature** – radius of curvature

radius ($t: float$) → float

Get radius of circle at distance t .

tangent ($t: float$) → Vec3

Get tangent at distance t as :class:`Vec3` object.

distance ($radius: float$) → float

Get distance L from origin for *radius*.

point ($t: float$) → Vec3

Get point at distance t as :class:`Vec3`.

circle_center ($t: float$) → Vec3

Get circle center at distance t .

approximate ($length: float, segments: int$) → Iterable[Vec3]

Approximate curve of length with line segments. Generates segments+1 vertices as *Vec3* objects.

bspline ($length: float, segments: int = 10, degree: int = 3, method: str = 'uniform'$) → BSpline

Approximate euler spiral as B-spline.

Parameters

- **length** – length of euler spiral

- **segments** – count of fit points for B-spline calculation
- **degree** – degree of BSpline
- **method** – calculation method for parameter vector t

Returns *BSpline*

Linear Algebra

Functions

`ezdxf.math.gauss_jordan_solver (A: Iterable[Iterable[float]], B: Iterable[Iterable[float]]) → Tuple[Matrix, Matrix]`

Solves the linear equation system given by a nxn Matrix A . x = B, right-hand side quantities as nxm Matrix B by the [Gauss-Jordan](#) algorithm, which is the slowest of all, but it is very reliable. Returns a copy of the modified input matrix A and the result matrix x.

Internally used for matrix inverse calculation.

Parameters

- **A** – matrix [[a11, a12, ..., a1n], [a21, a22, ..., a2n], [a21, a22, ..., a2n], ... [an1, an2, ..., ann]]
- **B** – matrix [[b11, b12, ..., b1m], [b21, b22, ..., b2m], ... [bn1, bn2, ..., bnm]]

Returns 2-tuple of *Matrix* objects

Raises `ZeroDivisionError` – singular matrix

`ezdxf.math.gauss_jordan_inverse (A: Iterable[Iterable[float]]) → Matrix`

Returns the inverse of matrix A as *Matrix* object.

Hint: For small matrices (n<10) is this function faster than `LUDecomposition(m).inverse()` and as fast even if the decomposition is already done.

Raises `ZeroDivisionError` – singular matrix

`ezdxf.math.gauss_vector_solver (A: Iterable[Iterable[float]], B: Iterable[float]) → List[float]`

Solves the linear equation system given by a nxn Matrix A . x = B, right-hand side quantities as vector B with n elements by the [Gauss-Elimination](#) algorithm, which is faster than the [Gauss-Jordan](#) algorithm. The speed improvement is more significant for solving multiple right-hand side quantities as matrix at once.

Reference implementation for error checking.

Parameters

- **A** – matrix [[a11, a12, ..., a1n], [a21, a22, ..., a2n], [a21, a22, ..., a2n], ... [an1, an2, ..., ann]]
- **B** – vector [b1, b2, ..., bn]

Returns vector as list of floats

Raises `ZeroDivisionError` – singular matrix

`ezdxf.math.gauss_matrix_solver(A: Iterable[Iterable[float]], B: Iterable[Iterable[float]]) → Matrix`

Solves the linear equation system given by a nxn Matrix A . x = B, right-hand side quantities as nxm Matrix B by the [Gauss-Elimination](#) algorithm, which is faster than the [Gauss-Jordan](#) algorithm.

Reference implementation for error checking.

Parameters

- **A** – matrix [[a11, a12, ..., a1n], [a21, a22, ..., a2n], [a21, a22, ..., a2n], ... [an1, an2, ..., ann]]
- **B** – matrix [[b11, b12, ..., b1m], [b21, b22, ..., b2m], ... [bn1, bn2, ..., bnm]]

Returns matrix as [`Matrix`](#) object

Raises `ZeroDivisionError` – singular matrix

`ezdxf.math.tridiagonal_vector_solver(A: Iterable[Iterable[float]], B: Iterable[float]) → List[float]`

Solves the linear equation system given by a tri-diagonal nxn Matrix A . x = B, right-hand side quantities as vector B. Matrix A is diagonal matrix defined by 3 diagonals [-1 (a), 0 (b), +1 (c)].

Note: a0 is not used but has to be present, cn-1 is also not used and must not be present.

If an `ZeroDivisionError` exception occurs, the equation system can possibly be solved by `BandedMatrixLU(A, 1, 1).solve_vector(B)`

Parameters

- **A** – diagonal matrix [[a0..an-1], [b0..bn-1], [c0..cn-1]]

```
[ [b0, c0, 0, 0, ...],
  [a1, b1, c1, 0, ...],
  [0, a2, b2, c2, ...],
  ... ]
```
- **B** – iterable of floats [[b1, b1, ..., bn]]

Returns list of floats

Raises `ZeroDivisionError` – singular matrix

`ezdxf.math.tridiagonal_matrix_solver(A: Iterable[Iterable[float]], B: Iterable[Iterable[float]]) → Matrix`

Solves the linear equation system given by a tri-diagonal nxn Matrix A . x = B, right-hand side quantities as n xm Matrix B. Matrix A is diagonal matrix defined by 3 diagonals [-1 (a), 0 (b), +1 (c)].

Note: a0 is not used but has to be present, cn-1 is also not used and must not be present.

If an `ZeroDivisionError` exception occurs, the equation system can possibly be solved by `BandedMatrixLU(A, 1, 1).solve_vector(B)`

Parameters

- **A** – diagonal matrix [[a0..an-1], [b0..bn-1], [c0..cn-1]]

```
[ [b0, c0, 0, 0, ...],
  [a1, b1, c1, 0, ...],
  [0, a2, b2, c2, ...],
  ... ]
```
- **B** – matrix [[b11, b12, ..., b1m], [b21, b22, ..., b2m], ... [bn1, bn2, ..., bnm]]

Returns matrix as [`Matrix`](#) object

Raises ZeroDivisionError – singular matrix

`ezdxf.math.banded_matrix(A: Matrix, check_all=True) → Tuple[int, int]`

Transform matrix A into a compact banded matrix representation. Returns compact representation as `Matrix` object and lower- and upper band count m1 and m2.

Parameters

- `A` – input `Matrix`
- `check_all` – check all diagonals if True or abort testing after first all zero diagonal if False.

`ezdxf.math.detect_banded_matrix(A: Matrix, check_all=True) → Tuple[int, int]`

Returns lower- and upper band count m1 and m2.

Parameters

- `A` – input `Matrix`
- `check_all` – check all diagonals if True or abort testing after first all zero diagonal if False.

`ezdxf.math.compact_banded_matrix(A: Matrix, m1: int, m2: int) → Matrix`

Returns compact banded matrix representation as `Matrix` object.

Parameters

- `A` – matrix to transform
- `m1` – lower band count, excluding main matrix diagonal
- `m2` – upper band count, excluding main matrix diagonal

`ezdxf.math.freeze_matrix(A: Union[MatrixData, Matrix]) → Matrix`

Returns a frozen matrix, all data is stored in immutable tuples.

Matrix Class

```
class eздxf.math.Matrix(items: Any = None, shape: Tuple[int, int] = None, matrix: List[List[float]] = None)
```

Basic matrix implementation without any optimization for speed of memory usage. Matrix data is stored in row major order, this means in a list of rows, where each row is a list of floats. Direct access to the data is accessible by the attribute `Matrix.matrix`.

The matrix can be frozen by function `freeze_matrix()` or method `Matrix.freeze()`, than the data is stored in immutable tuples.

Initialization:

- `Matrix(shape=(rows, cols))` ... new matrix filled with zeros
- `Matrix(matrix[, shape=(rows, cols)])` ... from copy of matrix and optional reshape
- `Matrix([[row_0], [row_1], ..., [row_n]])` ... from `Iterable[Iterable[float]]`
- `Matrix([a1, a2, ..., an], shape=(rows, cols))` ... from `Iterable[float]` and shape

nrows

Count of matrix rows.

ncols

Count of matrix columns.

shape

Shape of matrix as (n, m) tuple for n rows and m columns.

static reshape (items: Iterable[float], shape: Tuple[int, int]) → eздxf.math.linalg.Matrix

Returns a new matrix for iterable *items* in the configuration of *shape*.

classmethod identity (shape: Tuple[int, int]) → eздxf.math.linalg.Matrix

Returns the identity matrix for configuration *shape*.

row (index: int) → List[float]

Returns row *index* as list of floats.

iter_row (index: int) → Iterator[float]

Yield values of row *index*.

col (index: int) → List[float]

Return column *index* as list of floats.

iter_col (index: int) → Iterator[float]

Yield values of column *index*.

diag (index: int) → List[float]

Returns diagonal *index* as list of floats.

An *index* of 0 specifies the main diagonal, negative values specifies diagonals below the main diagonal and positive values specifies diagonals above the main diagonal.

e.g. given a 4x4 matrix:

- index 0 is [00, 11, 22, 33],
- index -1 is [10, 21, 32] and
- index +1 is [01, 12, 23]

iter_diag (index: int) → Iterator[float]

Yield values of diagonal *index*, see also [diag \(\)](#).

rows () → List[List[float]]

Return a list of all rows.

cols () → List[List[float]]

Return a list of all columns.

set_row (index: int, items: Union[float, Sequence[float]] = 1.0) → None

Set row values to a fixed value or from an iterable of floats.

set_col (index: int, items: Union[float, Iterable[float]] = 1.0) → None

Set column values to a fixed value or from an iterable of floats.

set_diag (index: int = 0, items: Union[float, Iterable[float]] = 1.0) → None

Set diagonal values to a fixed value or from an iterable of floats.

An *index* of 0 specifies the main diagonal, negative values specifies diagonals below the main diagonal and positive values specifies diagonals above the main diagonal.

e.g. given a 4x4 matrix: index 0 is [00, 11, 22, 33], index -1 is [10, 21, 32] and index +1 is [01, 12, 23]

append_row (items: Sequence[float]) → None

Append a row to the matrix.

append_col (items: Sequence[float]) → None

Append a column to the matrix.

swap_rows (*a*: int, *b*: int) → None
Swap rows *a* and *b* inplace.

swap_cols (*a*: int, *b*: int) → None
Swap columns *a* and *b* inplace.

transpose() → Matrix
Returns a new transposed matrix.

inverse() → Matrix
Returns inverse of matrix as new object.

determinant() → float
Returns determinant of matrix, raises `ZeroDivisionError` if matrix is singular.

freeze() → Matrix
Returns a frozen matrix, all data is stored in immutable tuples.

lu_decomp() → LUDecomposition
Returns the `LU decomposition` as `LUdecomposition` object, a faster linear equation solver.

__getitem__(*item*: Tuple[int, int]) → float
Get value by (row, col) index tuple, fancy slicing as known from numpy is not supported.

__setitem__(*item*: Tuple[int, int], *value*: float)
Set value by (row, col) index tuple, fancy slicing as known from numpy is not supported.

__eq__(*other*: Matrix) → bool
Returns True if matrices are equal, tolerance value for comparison is adjustable by the attribute `Matrix.abs_tol`.

__add__(*other*: Union[Matrix, float]) → Matrix
Matrix addition by another matrix or a float, returns a new matrix.

__sub__(*other*: Union[Matrix, float]) → Matrix
Matrix subtraction by another matrix or a float, returns a new matrix.

__mul__(*other*: Union[Matrix, float]) → Matrix
Matrix multiplication by another matrix or a float, returns a new matrix.

LUDecomposition Class

class `ezdxf.math.LUDecomposition`(*A*: Iterable[Iterable[float]])
Represents a `LU decomposition` matrix of *A*, raise `ZeroDivisionError` for a singular matrix.

This algorithm is a little bit faster than the `Gauss-Elimination` algorithm using CPython and much faster when using pypy.

The `LUDecomposition.matrix` attribute gives access to the matrix data as list of rows like in the `Matrix` class, and the `LUDecomposition.index` attribute gives access to the swapped row indices.

Parameters **A** – matrix [[*a*11, *a*12, ..., *a*1*n*], [*a*21, *a*22, ..., *a*2*n*], [*a*21, *a*22, ..., *a*2*n*], ... [*a**n*1, *a**n*2, ..., *a**n**n*]]

Raises `ZeroDivisionError` – singular matrix

nrows
Count of matrix rows (and cols).

solve_vector(*B*: Iterable[float]) → List[float]
Solves the linear equation system given by the nxn Matrix *A* . *x* = *B*, right-hand side quantities as vector *B* with *n* elements.

Parameters **B** – vector [b1, b2, ..., bn]

Returns vector as list of floats

solve_matrix (*B*: Iterable[Iterable[float]]) → Matrix

Solves the linear equation system given by the nxn Matrix A . x = B, right-hand side quantities as nxm Matrix B.

Parameters **B** – matrix [[b11, b12, ..., b1m], [b21, b22, ..., b2m], ... [bn1, bn2, ..., bnm]]

Returns matrix as *Matrix* object

inverse () → Matrix

Returns the inverse of matrix as *Matrix* object, raise `ZeroDivisionError` for a singular matrix.

determinant () → float

Returns the determinant of matrix, raises `ZeroDivisionError` if matrix is singular.

BandedMatrixLU Class

class eздxf.math.BandedMatrixLU (*A*: eздxf.math.linalg.Matrix, *m1*: int, *m2*: int)

Represents a LU decomposition of a compact banded matrix.

upper

Upper triangle

lower

Lower triangle

m1

Lower band count, excluding main matrix diagonal

m2

Upper band count, excluding main matrix diagonal

index

Swapped indices

nrows

Count of matrix rows.

solve_vector (*B*: Iterable[float]) → List[float]

Solves the linear equation system given by the banded nxn Matrix A . x = B, right-hand side quantities as vector B with n elements.

Parameters **B** – vector [b1, b2, ..., bn]

Returns vector as list of floats

solve_matrix (*B*: Iterable[Iterable[float]]) → Matrix

Solves the linear equation system given by the banded nxn Matrix A . x = B, right-hand side quantities as nxm Matrix B.

Parameters **B** – matrix [[b11, b12, ..., b1m], [b21, b22, ..., b2m], ... [bn1, bn2, ..., bnm]]

Returns matrix as *Matrix* object

determinant () → float

Returns the determinant of matrix.

6.8.8 Construction

Path

This module implements a geometric `Path`, supported by several render backends, with the goal to create such paths from DXF entities like LWPOLYLINE, POLYLINE or HATCH and send them to the render backend, see `ezdxf.addons.drawing`.

Minimum common interface:

- **matplotlib: PathPatch**
 - `matplotlib.path.Path()` codes:
 - `MOVETO`
 - `LINETO`
 - `CURVE3` - quadratic Bézier-curve
 - `CURVE4` - cubic Bézier-curve
- **PyQt: QPainterPath**
 - `moveTo()`
 - `lineTo()`
 - `quadTo()` - quadratic Bézier-curve (converted to a cubic Bézier-curve)
 - `cubicTo()` - cubic Bézier-curve
- **PyCairo: Context**
 - `move_to()`
 - `line_to()`
 - no support for quadratic Bézier-curve
 - `curve_to()` - cubic Bézier-curve
- **SVG: SVG-Path**
 - “M” - absolute move to
 - “L” - absolute line to
 - “Q” - absolute quadratic Bézier-curve
 - “C” - absolute cubic Bézier-curve

ARC and ELLIPSE entities are approximated by multiple cubic Bézier-curves, which are close enough for display rendering. Non-rational SPLINES of 3rd degree can be represented exact as multiple cubic Bézier-curves, other B-splines will be approximated. The XLINE and the RAY entities are not supported, because of their infinite nature.

This `Path` class is a full featured 3D object, although the backends only support 2D paths.

Hint: A `Path` can not represent a point. A `Path` with only a start point yields no vertices!

Changed in version 0.16: Refactored the module `ezdxf.render.path` into the subpackage `ezdxf.path`.

The usability of the `Path` class expanded by the introduction of the reverse conversion from `Path` to DXF entities (LWPOLYLINE, POLYLINE, LINE), and many other tools in `ezdxf` v0.16. To emphasize this new usability, the `Path` class has got its own subpackage `ezdxf.path`.

New in version 0.17: Added the `Path.move_to()` command and *Multi-Path* support.

Empty-Path Contains only a start point, the length of the path is 0 and the methods `Path.approximate()`, `Path.flattening()` and `Path.control_vertices()` do not yield any vertices.

Single-Path The `Path` object contains only one path without gaps, the property `Path.has_sub_paths` is `False` and the method `Path.sub_paths()` yields only this one path.

Multi-Path The `Path` object contains more than one path, the property `Path.has_sub_paths` is `True` and the method `Path.sub_paths()` yields all paths within this object as single-path objects. It is not possible to detect the orientation of a multi-path object, therefore the methods `Path.has_clockwise_orientation()`, `Path.clockwise()` and `Path.counter_clockwise()` raise a `TypeError` exception.

Warning: Always import from the top level `ezdxf.path`, never from the sub-modules

Factory Functions

Functions to create `Path` objects from other objects.

`ezdxf.path.make_path(entity: DXFEntity) → Path`

Factory function to create a single `Path` object from a DXF entity. Supported DXF types:

- LINE
- CIRCLE
- ARC
- ELLIPSE
- SPLINE and HELIX
- LWPOLYLINE
- 2D and 3D POLYLINE
- SOLID, TRACE, 3DFACE
- IMAGE, WIPEOUT clipping path
- VIEWPORT clipping path
- HATCH as *Multi-Path* object, new in v0.17

Parameters

- `entity` – DXF entity
- `segments` – minimal count of cubic Bézier-curves for elliptical arcs like CIRCLE, ARC, ELLIPSE, see `Path.add_ellipse()`
- `level` – subdivide level for SPLINE approximation, see `Path.add_spline()`

Raises `TypeError` – for unsupported DXF types

New in version 0.16.

Changed in version 0.17: support for HATCH as *Multi-Path* object

`ezdxf.path.from_hatch(hatch: Hatch) → Iterable[Path]`
Yield all HATCH boundary paths as separated `Path` objects.

New in version 0.16.

Changed in version 17.1: Attaches the boundary state to each path as `ezdxf.lldxf.const.BoundaryPathState`.

`ezdxf.path.from_vertices(vertices: Iterable[Vertex], close=False) → Path`
Returns a `Path` object from the given `vertices`.

`ezdxf.path.from_matplotlib_path(mpath, curves=True) → Iterable[Path]`
Yields multiple `Path` objects from a Matplotlib Path (TextPath) object. (requires Matplotlib)

New in version 0.16.

`ezdxf.path.multi_path_from_matplotlib_path(mpath, curves=True) → Path`
Returns a `Path` object from a Matplotlib Path (TextPath) object. (requires Matplotlib). Returns a multi-path object if necessary.

New in version 0.17.

`ezdxf.path.from_qpainter_path(qpath) → Iterable[Path]`
Yields multiple `Path` objects from a QPainterPath. (requires Qt bindings)

New in version 0.16.

`ezdxf.path.multi_path_from_qpainter_path(qpath) → Path`
Returns a `Path` objects from a QPainterPath. Returns a multi-path object if necessary. (requires Qt bindings)

New in version 0.17.

Render Functions

Functions to create DXF entities from paths and add them to the modelspace, a paperspace layout or a block definition.

`ezdxf.path.render_lwpolylines(layout: Layout, paths: Iterable[Path], *, distance: float = 0.01, segments: int = 4, extrusion: Vertex = (0, 0, 1), dxfattribs: Dict = None) → EntityQuery`

Render the given `paths` into `layout` as `LWPolyline` entities. The `extrusion` vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

Parameters

- `layout` – the modelspace, a paperspace layout or a block definition
- `paths` – iterable of `Path` objects
- `distance` – maximum distance, see `Path.flattening()`
- `segments` – minimum segment count per Bézier curve
- `extrusion` – extrusion vector for all paths
- `dfattribs` – additional DXF attribs

`Returns` created entities in an `EntityQuery` object

New in version 0.16.

```
ezdxf.path.render_polylines2d(layout: Layout, paths: Iterable[Path], *, distance: float = 0.01,
                               segments: int = 4, extrusion: Vertex = (0, 0, 1), dxfsattribs: Dict =
                               None) → EntityQuery
```

Render the given *paths* into *layout* as 2D [Polyline](#) entities. The *extrusion* vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

Parameters

- **layout** – the modelspace, a paperspace layout or a block definition
- **paths** – iterable of [Path](#) objects
- **distance** – maximum distance, see [Path.flattening\(\)](#)
- **segments** – minimum segment count per Bézier curve
- **extrusion** – extrusion vector for all paths
- **dxfsattribs** – additional DXF attribs

Returns created entities in an [EntityQuery](#) object

New in version 0.16.

```
ezdxf.path.render_hatches(layout: Layout, paths: Iterable[Path], *, edge_path = True, distance:
                           float = 0.01, segments: int = 4, g1_tol: float = 1e-4, extrusion: Vertex =
                           (0, 0, 1), dxfsattribs: Dict = None) → EntityQuery
```

Render the given *paths* into *layout* as [Hatch](#) entities. The *extrusion* vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

Parameters

- **layout** – the modelspace, a paperspace layout or a block definition
- **paths** – iterable of [Path](#) objects
- **edge_path** – True for edge paths build of LINE and SPLINE edges, False for only LWPOLYLINE paths as boundary paths
- **distance** – maximum distance, see [Path.flattening\(\)](#)
- **segments** – minimum segment count per Bézier curve to flatten polyline paths
- **g1_tol** – tolerance for G1 continuity check to separate SPLINE edges
- **extrusion** – extrusion vector for all paths
- **dxfsattribs** – additional DXF attribs

Returns created entities in an [EntityQuery](#) object

New in version 0.16.

```
ezdxf.path.render_mpolygons(layout: Layout, paths: Iterable[Path], *, distance: float = 0.01, seg-
                           ments: int = 4, extrusion: Vertex = (0, 0, 1), dxfsattribs: Dict = None)
                           → EntityQuery
```

Render the given *paths* into *layout* as [MPolygon](#) entities. The MPOLYGON entity supports only polyline boundary paths. All curves will be approximated.

The *extrusion* vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

Parameters

- **layout** – the modelspace, a paperspace layout or a block definition
- **paths** – iterable of `Path` objects
- **distance** – maximum distance, see `Path.flattening()`
- **segments** – minimum segment count per Bézier curve to flatten polyline paths
- **extrusion** – extrusion vector for all paths
- **dxfattribs** – additional DXF attribs

Returns created entities in an `EntityQuery` object

New in version 0.17.

```
ezdxf.path.render_polylines3d(layout: Layout, paths: Iterable[Path], *, distance: float = 0.01,
                               segments: int = 4, dxfattribs: Dict = None) → EntityQuery
```

Render the given `paths` into `layout` as 3D `Polyline` entities.

Parameters

- **layout** – the modelspace, a paperspace layout or a block definition
- **paths** – iterable of `Path` objects
- **distance** – maximum distance, see `Path.flattening()`
- **segments** – minimum segment count per Bézier curve
- **dxfattribs** – additional DXF attribs

Returns created entities in an `EntityQuery` object

New in version 0.16.

```
ezdxf.path.render_lines(layout: Layout, paths: Iterable[Path], *, distance: float = 0.01, segments:
                        int = 4, dxfattribs: Dict = None) → EntityQuery
```

Render the given `paths` into `layout` as `Line` entities.

Parameters

- **layout** – the modelspace, a paperspace layout or a block definition
- **paths** – iterable of `Path` objects
- **distance** – maximum distance, see `Path.flattening()`
- **segments** – minimum segment count per Bézier curve
- **dxfattribs** – additional DXF attribs

Returns created entities in an `EntityQuery` object

New in version 0.16.

```
ezdxf.path.render_splines_and_polylines(layout: Layout, paths: Iterable[Path], *, g1_tol:
                                         float = 1e-4, dxfattribs: Dict = None) → Entity-
                                         Query
```

Render the given `paths` into `layout` as `Spline` and 3D `Polyline` entities.

Parameters

- **layout** – the modelspace, a paperspace layout or a block definition
- **paths** – iterable of `Path` objects
- **g1_tol** – tolerance for G1 continuity check
- **dxfattribs** – additional DXF attribs

Returns created entities in an `EntityQuery` object

New in version 0.16.

Entity Maker

Functions to create DXF entities from paths.

```
ezdxf.path.to_lwpolylines(paths: Iterable[Path], *, distance: float = 0.01, segments: int = 4, extrusion: Vertex = (0, 0, 1), dxfattribs: Dict = None) → Iterable[LWPolyline]
```

Convert the given `paths` into `LWPolyline` entities. The `extrusion` vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

Parameters

- `paths` – iterable of `Path` objects
- `distance` – maximum distance, see `Path.flattening()`
- `segments` – minimum segment count per Bézier curve
- `extrusion` – extrusion vector for all paths
- `dxfattribs` – additional DXF attribs

Returns iterable of `LWPolyline` objects

New in version 0.16.

```
ezdxf.path.to_polylines2d(paths: Iterable[Path], *, distance: float = 0.01, segments: int = 4, extrusion: Vertex = (0, 0, 1), dxfattribs: Dict = None) → Iterable[Polyline]
```

Convert the given `paths` into 2D `Polyline` entities. The `extrusion` vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

Parameters

- `paths` – iterable of `Path` objects
- `distance` – maximum distance, see `Path.flattening()`
- `segments` – minimum segment count per Bézier curve
- `extrusion` – extrusion vector for all paths
- `dxfattribs` – additional DXF attribs

Returns iterable of 2D `Polyline` objects

New in version 0.16.

```
ezdxf.path.to_hatches(paths: Iterable[Path], *, edge_path: True, distance: float = 0.01, segments: int = 4, g1_tol: float = 1e-4, extrusion: Vertex = (0, 0, 1), dxfattribs: Dict = None) → Iterable[Hatch]
```

Convert the given `paths` into `Hatch` entities. Uses LWPOLYLINE paths for boundaries without curves and edge paths, build of LINE and SPLINE edges, as boundary paths for boundaries including curves. The `extrusion` vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

Parameters

- **paths** – iterable of [Path](#) objects
- **edge_path** – True for edge paths build of LINE and SPLINE edges, False for only LWPOLYLINE paths as boundary paths
- **distance** – maximum distance, see [Path.flattening\(\)](#)
- **segments** – minimum segment count per Bézier curve to flatten LWPOLYLINE paths
- **g1_tol** – tolerance for G1 continuity check to separate SPLINE edges
- **extrusion** – extrusion vector to all paths
- **dxfattribs** – additional DXF attribs

Returns iterable of [Hatch](#) objects

New in version 0.16.

`ezdxf.path.to_mpolygons(paths: Iterable[Path], *, distance: float = 0.01, segments: int = 4, extrusion: Vertex = (0, 0, 1), dxfattribs: Dict = None) → Iterable[MPolygon]`

Convert the given *paths* into [MPolygon](#) entities. In contrast to HATCH, MPOLYGON supports only polyline boundary paths. All curves will be approximated.

The *extrusion* vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The plane elevation is the distance from the WCS origin to the start point of the first path.

Parameters

- **paths** – iterable of [Path](#) objects
- **distance** – maximum distance, see [Path.flattening\(\)](#)
- **segments** – minimum segment count per Bézier curve to flatten LWPOLYLINE paths
- **extrusion** – extrusion vector to all paths
- **dxfattribs** – additional DXF attribs

Returns iterable of [MPolygon](#) objects

New in version 0.17.

`ezdxf.path.to_polylines3d(paths: Iterable[Path], *, distance: float = 0.01, segments: int = 4, dxfattribs: Dict = None) → Iterable[Polyline]`

Convert the given *paths* into 3D [Polyline](#) entities.

Parameters

- **paths** – iterable of [Path](#) objects
- **distance** – maximum distance, see [Path.flattening\(\)](#)
- **segments** – minimum segment count per Bézier curve
- **dxfattribs** – additional DXF attribs

Returns iterable of 3D [Polyline](#) objects

New in version 0.16.

`ezdxf.path.to_lines(paths: Iterable[Path], *, distance: float = 0.01, segments: int = 4, dxfattribs: Dict = None) → Iterable[Line]`

Convert the given *paths* into [Line](#) entities.

Parameters

- **paths** – iterable of [Path](#) objects

- **distance** – maximum distance, see `Path.flattening()`
- **segments** – minimum segment count per Bézier curve
- **dxfattribs** – additional DXF attribs

Returns iterable of `Line` objects

New in version 0.16.

```
ezdxf.path.to_splines_and_polylines(paths: Iterable[Path], *, g1_tol: float = 1e-4, dxfattribs: Dict = None) → Iterable[Union[Spline, Polyline]]
```

Convert the given `paths` into `Spline` and 3D `Polyline` entities.

Parameters

- **paths** – iterable of `Path` objects
- **g1_tol** – tolerance for G1 continuity check
- **dxfattribs** – additional DXF attribs

Returns iterable of `Line` objects

New in version 0.16.

Tool Maker

Functions to create construction tools.

```
ezdxf.path.to_bsplines_and_vertices(path: Path, g1_tol: float = 1e-4) → Iterable[Union[BSpline, List[Vec3]]]
```

Convert a `Path` object into multiple cubic B-splines and polylines as lists of vertices. Breaks adjacent Bézier without G1 continuity into separated B-splines.

Parameters

- **path** – `Path` objects
- **g1_tol** – tolerance for G1 continuity check

Returns `BSpline` and lists of `Vec3`

New in version 0.16.

```
ezdxf.path.to_matplotlib_path(paths: Iterable[Path], extrusion = (0, 0, 1)) → matplotlib.path.Path
```

Convert the given `paths` into a single `matplotlib.path.Path` object. The `extrusion` vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS z-axis. The Matplotlib `Path` is a 2D object with `OCS` coordinates and the z-elevation is lost. (requires Matplotlib)

Parameters

- **paths** – iterable of `Path` objects
- **extrusion** – extrusion vector for all paths

Returns `matplotlib Path` in OCS!

New in version 0.16.

```
ezdxf.path.to_qpainter_path(paths: Iterable[Path], extrusion = (0, 0, 1)) → QPainterPath
```

Convert the given `paths` into a `QtGui.QPainterPath` object. The `extrusion` vector is applied to all paths, all vertices are projected onto the plane normal to this extrusion vector. The default extrusion vector is the WCS

z-axis. The `QPainterPath` is a 2D object with `OCS` coordinates and the z-elevation is lost. (requires Qt bindings)

Parameters

- **paths** – iterable of `Path` objects
- **extrusion** – extrusion vector for all paths

Returns `QPainterPath` in OCS!

New in version 0.16.

Utility Functions

`ezdxf.path.transform_paths(paths: Iterable[Path], m: Matrix44) → List[Path]`

Transform multiple `Path` objects at once by transformation matrix `m`. Returns a list of the transformed `Path` objects. Warning: transformed paths loses the attached user data!

Parameters

- **paths** – iterable of `Path` objects
- **m** – transformation matrix of type `Matrix44`

`ezdxf.path.transform_paths_to_ocs(paths: Iterable[Path], ocs: OCS) → List[Path]`

Transform multiple `Path` objects at once from WCS to OCS. Returns a list of the transformed `Path` objects. Warning: transformed paths loses the attached user data!

Parameters

- **paths** – iterable of `Path` objects
- **ocs** – OCS transformation of type `OCS`

`ezdxf.path.bbox(paths: Iterable[Path]) → BoundingBox`

Returns the `BoundingBox` for the given paths.

Parameters

- **paths** – iterable of `Path` objects
- **flatten** – value != 0 for bounding box calculation from the flattened path and value == 0 for bounding box from the control vertices. Default value is 0.01 as max flattening distance.
- **segments** – minimal segment count for flattening

`ezdxf.path.fit_paths_into_box(paths: Iterable[Path], size: Tuple[float, float, float], uniform = True, source_box: BoundingBox = None) → List[Path]`

Scale the given `paths` to fit into a box of the given `size`, so that all path vertices are inside this borders. If `source_box` is `None` the default source bounding box is calculated from the control points of the `paths`.

Note: if the target size has a z-size of 0, the `paths` are projected into the xy-plane, same is true for the x-size, projects into the yz-plane and the y-size, projects into and xz-plane.

Parameters

- **paths** – iterable of `Path` objects
- **size** – target box size as tuple of x-, y- and z-size values
- **uniform** – `True` for uniform scaling
- **source_box** – pass precalculated source bounding box, or `None` to calculate the default source bounding box from the control vertices

`ezdxf.path.add_bezier3p(path: Path, curves: Iterable[Bezier3P])`

Add multiple quadratic Bézier-curves to the given `path`.

Auto-detect the connection point to the given `path`, if neither the start- nor the end point of the curves is close to the path end point, a line from the path end point to the start point of the first curve will be added automatically.

Changed in version 0.16.2: add linear Bézier curve segments as LINE_TO commands

`ezdxf.path.add_bezier4p(path: Path, curves: Iterable[Bezier4P])`

Add multiple cubic Bézier-curves to the given `path`.

Auto-detect the connection point to the given `path`, if neither the start- nor the end point of the curves is close to the path end point, a line from the path end point to the start point of the first curve will be added automatically.

Changed in version 0.16.2: add linear Bézier curve segments as LINE_TO commands

`ezdxf.path.add_ellipse(path: Path, ellipse: ConstructionEllipse, segments=1)`

Add an elliptical arc as multiple cubic Bézier-curves to the given `path`, use `from_arc()` constructor of class `ConstructionEllipse` to add circular arcs.

Auto-detect the connection point to the given `path`, if neither the start- nor the end point of the ellipse is close to the path end point, a line from the path end point to the ellipse start point will be added automatically (see `add_bezier4p()`).

By default the start of an **empty** path is set to the start point of the ellipse, setting argument `reset` to `False` prevents this behavior.

Parameters

- `path` – `Path` object
- `ellipse` – ellipse parameters as `ConstructionEllipse` object
- `segments` – count of Bézier-curve segments, at least one segment for each quarter ($\pi/2$), 1 for as few as possible.
- `reset` – set start point to start of ellipse if path is empty

`ezdxf.path.add_spline(path: Path, spline: BSpline, level=4)`

Add a B-spline as multiple cubic Bézier-curves.

Non-rational B-splines of 3rd degree gets a perfect conversion to cubic bezier curves with a minimal count of curve segments, all other B-spline require much more curve segments for approximation.

Auto-detect the connection point to the given `path`, if neither the start- nor the end point of the B-spline is close to the path end point, a line from the path end point to the start point of the B-spline will be added automatically. (see `add_bezier4p()`).

By default the start of an **empty** path is set to the start point of the spline, setting argument `reset` to `False` prevents this behavior.

Parameters

- `path` – `Path` object
- `spline` – B-spline parameters as `BSpline` object
- `level` – subdivision level of approximation segments
- `reset` – set start point to start of spline if path is empty

`ezdxf.path.to_multi_path(paths: Iterable[Path]) → Path`

Returns a multi-path object from all given paths and their sub-paths. Ignores paths without any commands (empty paths).

New in version 0.17.

`ezdxf.path.single_paths(paths: Iterable[Path]) → Iterable[Path]`

Yields all given paths and their sub-paths as single path objects.

New in version 0.17.

`ezdxf.path.have_close_control_vertices(a: Path, b: Path, *, rel_tol=1e-9, abs_tol=1e-12) → bool`

Returns True if the control vertices of given paths are close.

New in version 0.16.5.

Basic Shapes

`ezdxf.path.unit_circle(start_angle: float = 0, end_angle: float = 2π, segments: int = 1, transform: Matrix44 = None) → Path`

Returns an unit circle as a `Path` object, with the center at (0, 0, 0) and the radius of 1 drawing unit.

The arc spans from the start- to the end angle in counter clockwise orientation. The end angle has to be greater than the start angle and the angle span has to be greater than 0.

Parameters

- **start_angle** – start angle in radians
- **end_angle** – end angle in radians ($\text{end_angle} > \text{start_angle!}$)
- **segments** – count of Bézier-curve segments, default is one segment for each arc quarter ($\pi/2$)
- **transform** – transformation Matrix applied to the unit circle

`ezdxf.path.wedge(start_angle: float, end_angle: float, segments: int = 1, transform: Matrix44 = None) → Path`

Returns a wedge as a `Path` object, with the center at (0, 0, 0) and the radius of 1 drawing unit.

The arc spans from the start- to the end angle in counter clockwise orientation. The end angle has to be greater than the start angle and the angle span has to be greater than 0.

Parameters

- **start_angle** – start angle in radians
- **end_angle** – end angle in radians ($\text{end_angle} > \text{start_angle!}$)
- **segments** – count of Bézier-curve segments, default is one segment for each arc quarter ($\pi/2$)
- **transform** – transformation Matrix applied to the wedge

`ezdxf.path.elliptic_transformation(center: Vertex = (0, 0, 0), radius: float = 1, ratio: float = 1, rotation: float = 0) → Matrix44`

Returns the transformation matrix to transform an unit circle into an arbitrary circular- or elliptic arc.

Example how to create an ellipse with an major axis length of 3, a minor axis length 1.5 and rotated about 90°:

```
m = elliptic_transformation(radius=3, ratio=0.5, rotation=math.pi / 2)
ellipse = shapes.unit_circle(transform=m)
```

Parameters

- **center** – curve center in WCS
- **radius** – radius of the major axis in drawing units

- **ratio** – ratio of minor axis to major axis
- **rotation** – rotation angle about the z-axis in radians

`ezdxf.path.rect(width: float = 1, height: float = 1, transform: Matrix44 = None) → Path`

Returns a closed rectangle as a `Path` object, with the center at (0, 0, 0) and the given `width` and `height` in drawing units.

Parameters

- **width** – width of the rectangle in drawing units, `width > 0`
- **height** – height of the rectangle in drawing units, `height > 0`
- **transform** – transformation Matrix applied to the rectangle

`ezdxf.path.ngon(count: int, length: float = None, radius: float = 1.0, transform: Matrix44 = None) → Path`

Returns a regular polygon a `Path` object, with the center at (0, 0, 0). The polygon size is determined by the edge `length` or the circum `radius` argument. If both are given `length` has higher priority. Default size is a `radius` of 1. The ngon starts with the first vertex is on the x-axis! The base geometry is created by function `ezdxf.render.forms.ngon()`.

Parameters

- **count** – count of polygon corners ≥ 3
- **length** – length of polygon side
- **radius** – circum radius, default is 1
- **transform** – transformation Matrix applied to the ngon

`ezdxf.path.star(count: int, r1: float, r2: float, transform: Matrix44 = None) → Path`

Returns a star shape as a `Path` object, with the center at (0, 0, 0).

Argument `count` defines the count of star spikes, `r1` defines the radius of the “outer” vertices and `r2` defines the radius of the “inner” vertices, but this does not mean that `r1` has to be greater than `r2`. The star shape starts with the first vertex is on the x-axis! The base geometry is created by function `ezdxf.render.forms.star()`.

Parameters

- **count** – spike count ≥ 3
- **r1** – radius 1
- **r2** – radius 2
- **transform** – transformation Matrix applied to the star

`ezdxf.path.gear(count: int, top_width: float, bottom_width: float, height: float, outside_radius: float, transform: Matrix44 = None) → Path`

Returns a gear (cogwheel) shape as a `Path` object, with the center at (0, 0, 0). The base geometry is created by function `ezdxf.render.forms.gear()`.

Warning: This function does not create correct gears for mechanical engineering!

Parameters

- **count** – teeth count ≥ 3
- **top_width** – teeth width at outside radius
- **bottom_width** – teeth width at base radius

- **height** – teeth height; base radius = outside radius - height
- **outside_radius** – outside radius
- **transform** – transformation Matrix applied to the gear shape

The `text2path` add-on provides additional functions to create paths from text strings and DXF text entities.

The Path Class

`class ezdxf.path.Path`

sub_paths() → Iterable[Path]

Yield sub-path as *Single-Path* objects.

It is safe to call `sub_paths()` on any path-type: *Single-Path*, *Multi-Path* and *Empty-Path*.

New in version 0.17.

control_vertices()

Yields all path control vertices in consecutive order.

has_clockwise_orientation() → bool

Returns True if 2D path has clockwise orientation, ignores z-axis of all control vertices.

Raises `TypeError` – can't detect orientation of a *Multi-Path* object

line_to(location: Vec3)

Add a line from actual path end point to *location*.

move_to(location: Vec3)

Start a new sub-path at *location*. This creates a gap between the current end-point and the start-point of the new sub-path. This converts the instance into a *Multi-Path* object.

If the `move_to()` command is the first command, the start point of the path will be reset to *location*.

New in version 0.17.

curve3_to(location: Vec3, ctrl: Vec3)

Add a quadratic Bézier-curve from actual path end point to *location*, *ctrl* is the control point for the quadratic Bézier-curve.

curve4_to(location: Vec3, ctrl1: Vec3, ctrl2: Vec3)

Add a cubic Bézier-curve from actual path end point to *location*, *ctrl1* and *ctrl2* are the control points for the cubic Bézier-curve.

close() → None

Close path by adding a line segment from the end point to the start point.

close_sub_path() → None

Close last sub-path by adding a line segment from the end point to the start point of the last sub-path. Behaves like `close()` for *Single-Path* instances.

New in version 0.17.

clone() → Path

Returns a new copy of *Path* with shared immutable data.

reversed() → Path

Returns a new *Path* with reversed segments and control vertices.

clockwise() → Path

Returns new [Path](#) in clockwise orientation.

Raises `TypeError` – can't detect orientation of a [Multi-Path](#) object

counter_clockwise() → Path

Returns new [Path](#) in counter-clockwise orientation.

Raises `TypeError` – can't detect orientation of a [Multi-Path](#) object

transform(m: Matrix44) → Path

Returns a new transformed path.

Parameters `m` – transformation matrix of type [Matrix44](#)

approximate(segments: int=20) → Iterable[Vec3]

Approximate path by vertices, `segments` is the count of approximation segments for each Bézier curve.

Does not yield any vertices for empty paths, where only a start point is present!

Approximation of [Multi-Path](#) objects is possible, but gaps are indistinguishable from line segments.

flattening(distance: float, segments: int=16) → Iterable[Vec3]

Approximate path by vertices and use adaptive recursive flattening to approximate Bézier curves. The argument `segments` is the minimum count of approximation segments for each curve, if the distance from the center of the approximation segment to the curve is bigger than `distance` the segment will be subdivided.

Does not yield any vertices for empty paths, where only a start point is present!

Flattening of [Multi-Path](#) objects is possible, but gaps are indistinguishable from line segments.

Parameters

- **distance** – maximum distance from the center of the curve to the center of the line segment between two approximation points to determine if a segment should be subdivided.
- **segments** – minimum segment count per Bézier curve

append_path(path: Path)

Append another path to this path. Adds a `self.line_to(path.start)` if the end of this path != the start of appended path.

New in version 0.17.

extend_multi_path(path: Path)

Extend the path by another path. The source path is automatically a [Multi-Path](#) object, even if the previous end point matches the start point of the appended path. Ignores paths without any commands (empty paths).

New in version 0.17.

all_lines_to_curve3() → None

Inline conversion of all LINE_TO commands into CURVE3_TO commands.

all_lines_to_curve4() → None

Inline conversion of all LINE_TO commands into CURVE4_TO commands.

Disassemble

New in version 0.16. This module provide tools for the recursive decomposition of nested block reference structures into a flat stream of DXF entities and converting DXF entities into geometric primitives of [Path](#) and [MeshBuilder](#) objects encapsulated into intermediate [Primitive](#) classes.

Changed in version 0.17: The `Hatch` entity is no special case anymore and has regular support by the `make_primitive()` function.

Warning: Do not expect advanced vectorization capabilities: Text entities like TEXT, ATTRIB, ATTDEF and MTEXT get only a rough border box representation. The `text2path` add-on can convert text into paths. VIEWPORT, IMAGE and WIPEOUT are represented by their clipping path. Unsupported entities: all ACIS based entities, XREF, UNDERLAY, ACAD_TABLE, RAY, XLINE. Unsupported entities will be ignored.

Text Boundary Calculation

Text boundary calculations are based on monospaced (fixed-pitch, fixed-width, non-proportional) font metrics, which do not provide a good accuracy for text height calculation and much less accuracy for text width calculation. It is possible to improve this results by using the font support from the **optional Matplotlib** package.

Install Matplotlib from command line:

```
C:\> pip3 install matplotlib
```

The *Matplotlib* font support will improve the results for TEXT, ATTRIB and ATTDEF. The MTEXT entity has many advanced features which would require a full “Rich Text Format” rendering and that is far beyond the goals and capabilities of this library, therefore the boundary box for MTEXT will **never** be as accurate as in a dedicated CAD application.

Using the *Matplotlib* font support adds **runtime overhead**, therefore it is possible to deactivate the *Matplotlib* font support by setting the global option:

```
options.use_matplotlib_font_support = False
```

Flatten Complex DXF Entities

```
ezdxf.disassemble.recursive_decompose(entities: Iterable[DXFEntity]) → Iterable[DXFEntity]
```

Recursive decomposition of the given DXF entity collection into a flat DXF entity stream. All block references (INSERT) and entities which provide a `virtual_entities()` method will be disassembled into simple DXF sub-entities, therefore the returned entity stream does not contain any INSERT entity.

Point entities will **not** be disassembled into DXF sub-entities, as defined by the current point style \$PDMODE.

These entity types include sub-entities and will be decomposed into simple DXF entities:

- INSERT
- DIMENSION
- LEADER
- MLEADER
- MLINE

Decomposition of XREF, UNDERLAY and ACAD_TABLE entities is not supported.

Entity Deconstruction

These functions disassemble DXF entities into simple geometric objects like meshes, paths or vertices. The `Primitive` is a simplified intermediate class to use a common interface on various DXF entities.

`ezdxf.disassemble.make_primitive(entity: DXFEntity, max_flattening_distance=None) → Primitive`

Factory to create path/mesh primitives. The `max_flattening_distance` defines the max distance between the approximation line and the original curve. Use `max_flattening_distance` to override the default value.

Returns an **empty primitive** for unsupported entities. The `empty` state of a primitive can be checked by the property `is_empty`. The `path` and the `mesh` attributes of an empty primitive are `None` and the `vertices()` method yields no vertices.

Changed in version 0.17: regular support for the `Hatch` entity.

`ezdxf.disassemble.to_primitives(entities: Iterable[DXFEntity], max_flattening_distance: float = None) → Iterable[Primitive]`

Yields all DXF entities as path or mesh primitives. Yields unsupported entities as empty primitives, see `make_primitive()`.

Parameters

- `entities` – iterable of DXF entities
- `max_flattening_distance` – override the default value

`ezdxf.disassemble.to_meshes(primitives: Iterable[Primitive]) → Iterable[MeshBuilder]`

Yields all `MeshBuilder` objects from the given `primitives`. Ignores primitives without a defined mesh.

`ezdxf.disassemble.to_paths(primitives: Iterable[Primitive]) → Iterable[Path]`

Yields all `Path` objects from the given `primitives`. Ignores primitives without a defined path.

`ezdxf.disassemble.to_vertices(primitives: Iterable[Primitive]) → Iterable[Vec3]`

Yields all vertices from the given `primitives`. Paths will be flattened to create the associated vertices. See also `to_control_vertices()` to collect only the control vertices from the paths without flattening.

`ezdxf.disassemble.to_control_vertices(primitives: Iterable[Primitive]) → Iterable[Vec3]`

Yields all path control vertices and all mesh vertices from the given `primitives`. Like `to_vertices()`, but without flattening.

class `ezdxf.disassemble.Primitive`

Interface class for path/mesh primitives.

entity

Reference to the source DXF entity of this primitive.

max_flattening_distance

The `max_flattening_distance` attribute defines the max distance in drawing units between the approximation line and the original curve. Set the value by direct attribute access. (float) default = 0.01

vertices() → Iterable[Vec3]

Yields all vertices of the path/mesh representation as `Vec3` objects.

Bounding Box

New in version 0.16. The `ezdxf.bbox` module provide tools to calculate bounding boxes for many DXF entities, but not for all. The bounding box calculation is based on the `ezdxf.disassemble` module and therefore has the same limitation.

Warning: If accurate boundary boxes for text entities are important for you, read this first: [Text Boundary Calculation](#). TL;DR: Boundary boxes for text entities are **not accurate!**

Unsupported DXF entities:

- All ACIS based types like BODY, 3DSOLID or REGION
- External references (XREF) and UNDERLAY object
- RAY and XRAY, extend into infinite
- ACAD_TABLE, no basic support - only preserved by *ezdxf*

Unsupported entities are silently ignored, filtering of these DXF types is not necessary.

The base type for bounding boxes is the `BoundingBox` class from the module `ezdxf.math`.

The *entities* iterable as input can be the whole modelspace, an entity query or any iterable container of DXF entities.

The Calculation of bounding boxes of curves is done by flattening the curve by a default flattening distance of 0.01. Set argument *flatten* to 0 to speedup the bounding box calculation by accepting less precision for curved objects by using only the control vertices.

The **optional** caching object `Cache` has to be instantiated by the user, this is only useful if the same entities will be processed multiple times.

Example usage with caching:

```
from ezdxf import bbox

msp = doc.modelspace()
cache = bbox.Cache()
# get overall bounding box
first_bbox = bbox.extents(msp, cache=cache)
# bounding box of all LINE entities
second_bbox = bbox.extend(msp.query("LINE"), cache=cache)
```

Functions

`ezdxf.bbox.extents(entities: Iterable[DXFEntity], *, flatten=0.01, cache: Cache=None) → BoundingBox`

Returns a single bounding box for all given *entities*.

Calculate bounding boxes from flattened curves, if argument *flatten* is not 0 (max flattening distance), else from control points.

`ezdxf.bbox.multi_flat(entities: Iterable[DXFEntity], *, flatten=0.01, cache: Cache=None) → Iterable[BoundingBox]`

Yields a bounding box for each of the given *entities*.

Calculate bounding boxes from flattened curves, if argument *flatten* is not 0 (max flattening distance), else from control points.

`ezdxf.bbox.multi_recursive(entities: Iterable[DXFEntity], *, flatten=0.01, cache: Cache=None) → Iterable[BoundingBox]`

Yields all bounding boxes for the given *entities* **or** all bounding boxes for their sub entities. If an entity (INSERT) has sub entities, only the bounding boxes of these sub entities will be yielded, **not** the bounding box of entity (INSERT) itself.

Calculate bounding boxes from flattened curves, if argument *flatten* is not 0 (max flattening distance), else from control points.

Caching Strategies

Because *ezdxf* is not a CAD application, *ezdxf* does not manage data structures which are optimized for a usage by a CAD kernel. This means that the content of complex entities like block references or leaders has to be created on demand by DXF primitives on the fly. These temporarily created entities are called virtual entities and have no handle and are not stored in the entity database.

All this is required to calculate the bounding box of complex entities, and it is therefore a very time consuming task. By using a *Cache* object it is possible to speedup this calculations, but this is not a magically feature, it requires an understanding of what is happening under the hood to achieve any performance gains.

For a single bounding box calculation, without any reuse of entities it makes no sense of using a *Cache* object, e.g. calculation of the modelspace extents:

```
from pathlib import Path
import eздxf
from eздxf import bbox

CADKitSamples = Path(eздxf.EZDXF_TEST_FILES) / 'CADKitSamples'

doc = eздxf.readfile(CADKitSamples / 'A_000217.dxf')
cache = bbox.Cache()
ext = bbox.extents(doc.modelspace(), cache)

print(cache)
```

1226 cached objects and not a single cache hit:

```
Cache(n=1226, hits=0, misses=3273)
```

The result for using UUIDs to cache virtual entities is not better:

```
Cache(n=2206, hits=0, misses=3273)
```

Same count of hits and misses, but now the cache also references ~1000 virtual entities, which block your memory until the cache is deleted, luckily this is a small DXF file (~838 kB).

Bounding box calculations for multiple entity queries, which have overlapping entity results, using a *Cache* object may speedup the calculation:

```
doc = eздxf.readfile(CADKitSamples / 'A_000217.dxf.dxf')
msp = doc.modelspace()
cache = bbox.Cache(uuid=False)

ext = bbox.extents(msp, cache)
print(cache)

# process modelspace again
ext = bbox.extents(msp, cache)
print(cache)
```

Processing the same data again leads some hits:

```
1st run: Cache(n=1226, hits=0, misses=3273)
2nd run: Cache(n=1226, hits=1224, misses=3309)
```

Using `uuid=True` leads not to more hits, but more cache entries:

```
1st run: Cache(n=2206, hits=0, misses=3273)
2nd run: Cache(n=2206, hits=1224, misses=3309)
```

Creating stable virtual entities by disassembling the entities at first leads to more hits:

```
from ezdxf import disassemble

entities = list(disassemble.recursive_decompose(msp))
cache = bbox.Cache(uuid=False)

bbox.extents(entities, cache)
print(cache)

bbox.extents(entities, cache)
print(cache)
```

First without UUID for stable virtual entities:

```
1st run: Cache(n=1037, hits=0, misses=4074)
2nd run: Cache(n=1037, hits=1037, misses=6078)
```

Using UUID for stable virtual entities leads to more hits:

```
1st run: Cache(n=2019, hits=0, misses=4074)
2nd run: Cache(n=2019, hits=2018, misses=4116)
```

But caching virtual entities needs also more memory.

In conclusion: Using a cache is only useful, if you often process **nearly the same data**; only then can an increase in performance be expected.

Cache Class

```
class ezdxf.bbox.Cache(uuid=False)
    Caching object for ezdxf.math.BoundingBox objects.
```

Parameters `uuid` – use UUIDs for virtual entities

`hits`

`misses`

`invalidate(entities: Iterable[DXFEntity]) → None`

Invalidate cache entries for the given DXF `entities`.

If entities are changed by the user, it is possible to invalidate individual entities. Use with care - discarding the whole cache is the safer workflow.

Ignores entities which are not stored in cache.

Upright

New in version 0.17. The functions in this module can help to convert an inverted *OCS* defined by an extrusion vector $(0, 0, -1)$ into a *WCS* aligned OCS defined by an extrusion vector $(0, 0, 1)$.

This simplifies 2D entity processing for *ezdxf* users and creates DXF output for 3rd party DXF libraries which ignore the existence of the *OCS*.

Supported DXF entities:

- CIRCLE
- ARC
- ELLIPSE (WCS entity, flips only the extrusion vector)
- SOLID
- TRACE
- LWPOLYLINE
- POLYLINE (only 2D entities)
- HATCH
- MPOLYGON
- INSERT (block references)

Warning: The WCS representation of OCS entities with flipped extrusion vector is not 100% identical to the source entity, curve orientation and vertex order may change, see *additional explanation* below. A mirrored text represented by an extrusion vector $(0, 0, -1)$ cannot be represented by an extrusion vector $(0, 0, 1)$, therefore this CANNOT work for text entities or entities including text: TEXT, ATTRIB, ATTDEF, MTEXT, DIMENSION, LEADER, MLEADER

Usage

The functions can be applied to any DXF entity without expecting errors or exceptions if the DXF entity is not supported or the extrusion vector differs from $(0, 0, -1)$. This also means you can apply the functions multiple times to the same entities without any problems. A common case would be to upright all entities of the model space:

```
import ezdxf
from ezdxf.upright import upright_all

doc = ezdxf.readfile("your.dxf")
msp = doc.modelspace()
upright_all(msp)
# doing it again is no problem but also has no further effects
upright_all(msp)
```

Another use case is exploding block references (INSERT) which may include reflections (= scaling by negative factors) that can lead to inverted extrusion vectors.

```
for block_ref in msp.query("INSERT"):
    entities = block_ref.explode()  # -> EntityQuery object
    upright_all(entities)
```

Functions

`ezdxf.upright.upright(entity: DXFGraphic) → None`

Flips an inverted *OCS* defined by extrusion vector (0, 0, -1) into a *WCS* aligned *OCS* defined by extrusion vector (0, 0, 1). DXF entities with other extrusion vectors and unsupported DXF entities will be silently ignored. For more information about the limitations read the documentation of the `ezdxf.upright` module.

`ezdxf.upright.upright_all(entities: Iterable[DXFGraphic]) → None`

Call function `upright()` for all DXF entities in iterable *entities*:

```
upright_all(doc.modelspace())
```

Additional Explanation

This example shows why the entities with an inverted OCS, extrusion vector is (0, 0, -1), are not exact the same as with an WCS aligned OCS, extrusion vector is (0, 0, 1).

Note: The ARC entity represents the curve **always** in counter-clockwise orientation around the extrusion vector.

```
import eздxf
from eздxf.upright import upright
from eздxf.math import Matrix44

doc = eздxf.new()
msp = doc.modelspace()

arc = msp.add_arc(
    (5, 0),
    radius=5,
    start_angle=-90,
    end_angle=90,
    dxftattribs={"color": eздxf.const.RED},
)
# draw lines to the start- and end point of the ARC
msp.add_line((0, 0), arc.start_point, dxftattribs={"color": eздxf.const.GREEN})
msp.add_line((0, 0), arc.end_point, dxftattribs={"color": eздxf.const.BLUE})

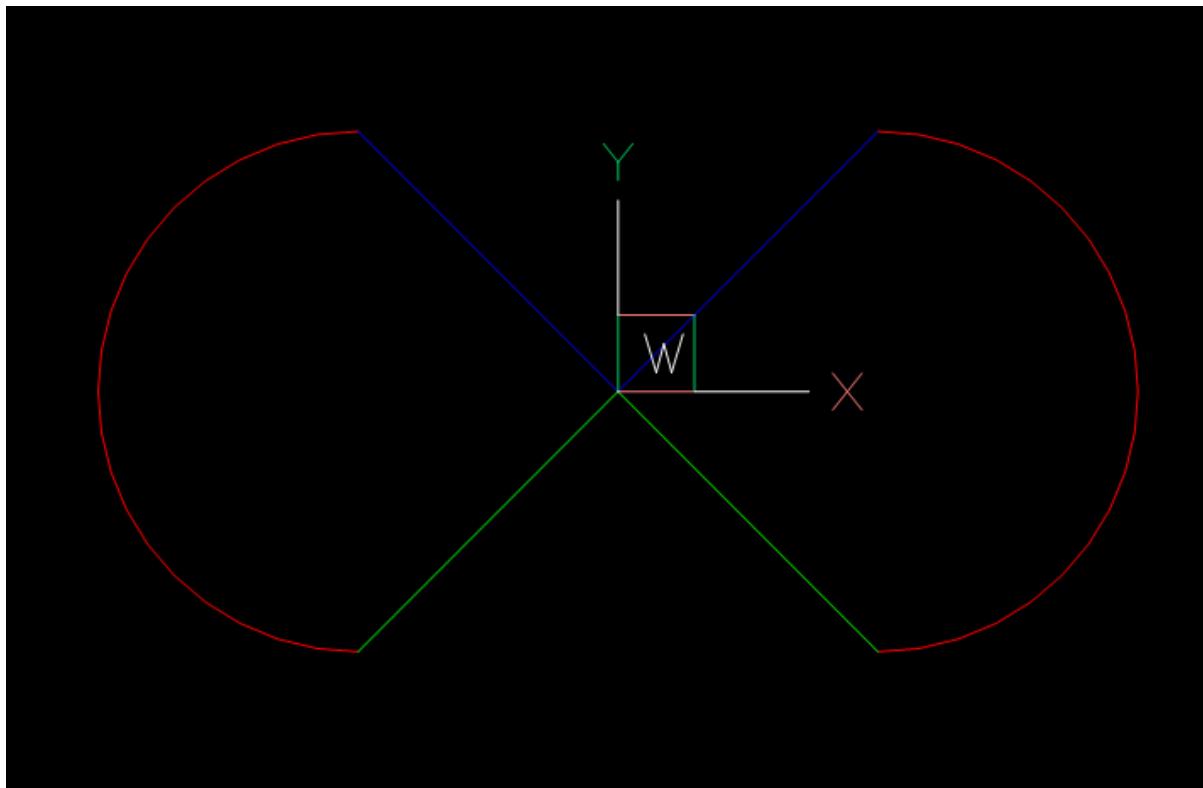
# copy arc
mirrored_arc = arc.copy()
msp.add_entity(mirrored_arc)

# mirror copy
mirrored_arc.transform(Matrix44.scale(-1, 1, 1))

# This creates an inverted extrusion vector:
assert mirrored_arc.dxf.extrusion.isclose((0, 0, -1))

# draw lines to the start- and end point of the mirrored ARC
msp.add_line((0, 0), mirrored_arc.start_point, dxftattribs={"color": eздxf.const.GREEN}
→)
msp.add_line((0, 0), mirrored_arc.end_point, dxftattribs={"color": eздxf.const.BLUE})
```

Result without applying the `upright()` function - true mirroring:



```
...
# This creates an inverted extrusion vector:
assert mirrored_arc.dxf.extrusion.isclose((0, 0, -1))

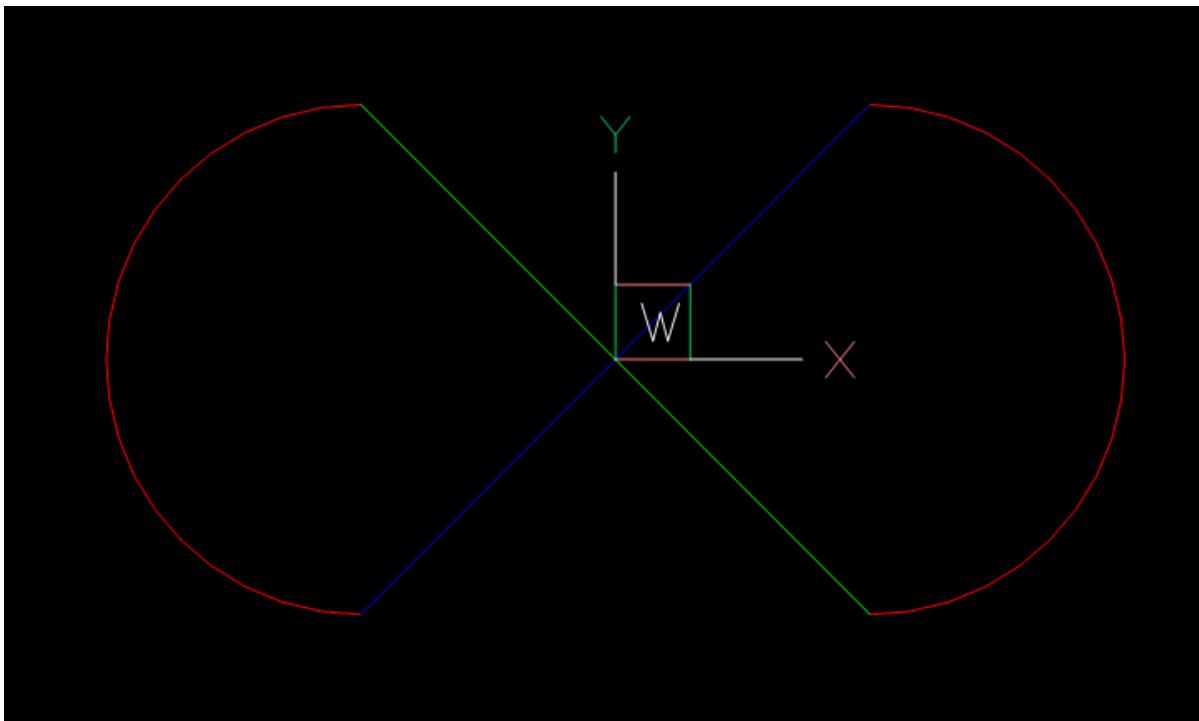
start_point_inv = mirrored_arc.start_point
end_point_inv = mirrored_arc.end_point

upright(mirrored_arc)
# OCS is aligned with WCS:
assert mirrored_arc.dxf.extrusion.isclose((0, 0, 1))

# start- and end points are swapped after applying upright()
assert mirrored_arc.start_point.isclose(end_point_inv)
assert mirrored_arc.end_point.isclose(start_point_inv)

# draw lines to the start- and end point of the mirrored ARC
msp.add_line((0, 0), mirrored_arc.start_point, dxfattribs={"color": ezdxf.const.GREEN}
             ↵)
msp.add_line((0, 0), mirrored_arc.end_point, dxfattribs={"color": ezdxf.const.BLUE})
```

Result after applying the `upright()` function - false mirroring:



To avoid this issue the ARC entity would have to represent the curve in clockwise orientation around the extrusion vector $(0, 0, 1)$, which is not possible!

Note: The shape of the mirrored arcs is the same for both extrusion vectors, but the start- and the end points are swapped (reversed vertex order)!

6.8.9 Custom Data

Custom XDATA

The classes `XDataUserList` and `XDataUserDict` manage custom user data stored in the XDATA section of a DXF entity. For more information about XDATA see reference section: [Extended Data \(XDATA\)](#)

These classes store only a limited set of data types with fixed group codes and the types are checked by `isinstance()` so a `Vec3` object can not be replaced by a (x, y, z) -tuple:

Group Code	Data Type
1000	str, limited to 255 characters, line breaks " <code>\n</code> " and " <code>\r</code> " are not allowed
1010	<code>Vec3</code>
1040	float
1071	32-bit int, restricted by the DXF standard not by Python!

Strings are limited to 255 characters, line breaks "`\n`" and "`\r`" are not allowed.

This classes assume a certain XDATA structure and therefore can not manage arbitrary XDATA!

This classes do not create the required AppID table entry, only the default AppID "EZDXF" exist by default. Setup a new AppID in the AppID table: `doc.appids.add("MYAPP")`.

For usage look at this [example](#) at github or go to the tutorial: [Storing Custom Data in DXF Files](#).

See also:

- Tutorial: [Storing Custom Data in DXF Files](#)
- Example at [github](#)
- XDATA reference: [Extended Data \(XDATA\)](#)
- XDATA management class: [XData](#)

XDataUserList

```
class ezdxf.entities.xdata.XDataUserList
```

Manage user data as a named list-like object in XDATA. Multiple user lists with different names can be stored in a single [XData](#) instance for a single AppID.

Recommended usage by context manager `entity()`:

```
with XDataUserList.entity(entity, name="MyList", appid="MYAPP") as ul:
    ul.append("The value of PI") # str "\n" and "\r" are not allowed
    ul.append(3.141592) # float
    ul.append(1) # int
    ul.append(Vec3(1, 2, 3)) # Vec3

    # invalid data type raises DXFTypeError
    ul.append((1, 2, 3)) # tuple instead of Vec3

    # retrieve a single value
    s = ul[0]

    # store whole content into a Python list
    data = list(ul)
```

Implements the `MutableSequence` interface.

xdata

The underlying [XData](#) instance.

```
__init__(xdata: ezdxf.entities.xdata.XData = None, name='DefaultList', appid='EZDXF')
```

Setup a XDATA user list `name` for the given `appid`.

The data is stored in the given `xdata` object, or in a new created [XData](#) instance if `None`. Changes of the content has to be committed at the end to be stored in the underlying `xdata` object.

Parameters

- `xdata` ([XData](#)) – underlying [XData](#) instance, if `None` a new one will be created
- `name` (`str`) – name of the user list
- `appid` (`str`) – application specific AppID

__str__()

Return `str(self)`.

__len__() → int

Returns `len(self)`.

__getitem__(item)

Get `self[item]`.

__setitem__(item, value)

Set self[item] to value.

__delitem__(item)

Delete self[item].

classmethod entity(entity: DXFEntity, name='DefaultList', appid='EZDXF') → Iterator[XDataUserList]

Context manager to manage a XDATA list *name* for a given DXF *entity*. Appends the user list to the existing *XData* instance or creates new *XData* instance.

Parameters

- **entity** (*DXFEntity*) – target DXF entity for the XDATA
- **name** (*str*) – name of the user list
- **appid** (*str*) – application specific AppID

commit() → None

Store all changes to the underlying *XData* instance. This call is not required if using the *entity()* context manager.

Raises

- *DXFValueError* – invalid chars "\n" or "\r" in a string
- *DXFTypError* – invalid data type

XDataUserDict**class ezdxf.entities.xdata.XDataUserDict**

Manage user data as a named dict-like object in XDATA. Multiple user dicts with different names can be stored in a single *XData* instance for a single AppID. The keys have to be strings.

Recommended usage by context manager *entity()*:

```
with XDataUserDict.entity(entity, name="MyDict", appid="MYAPP") as ud:
    ud["comment"] = "The value of PI" # str "\n" and "\r" are not allowed
    ud["pi"] = 3.141592 # float
    ud["number"] = 1 # int
    ud["vertex"] = Vec3(1, 2, 3) # Vec3

    # invalid data type raises DXFTypError
    ud["vertex"] = (1, 2, 3) # tuple instead of Vec3

    # retrieve single values
    s = ud["comment"]
    pi = ud.get("pi", 3.141592)

    # store whole content into a Python dict
    data = dict(ud)
```

Implements the *MutableMapping* interface.

The data is stored in XDATA like a *XDataUserList* by (key, value) pairs, therefore a *XDataUserDict* can also be loaded as *XDataUserList*. It is not possible to distinguish a *XDataUserDict* from a *XDataUserList* except by the name of the data structure.

xdata

The underlying *XData* instance.

__init__ (*xdata*: `ezdxf.entities.XData` = `None`, *name*=`'DefaultDict'`, *appid*=`'EZDXF'`)
Setup a XDATA user dict *name* for the given *appid*.

The data is stored in the given *xdata* object, or in a new created `XData` instance if `None`. Changes of the content has to be committed at the end to be stored in the underlying *xdata* object.

Parameters

- **xdata** (`XData`) – underlying `XData` instance, if `None` a new one will be created
- **name** (`str`) – name of the user list
- **appid** (`str`) – application specific AppID

__str__ ()

Return str(self).

__len__ ()

Returns len(self).

__getitem__ (*key*)

Get self[key].

__setitem__ (*key*, *item*)

Set self[key] to value, key has to be a string.

Raises `DXFTypeError` – key is not a string

__delitem__ (*key*)

Delete self[key].

discard (*key*)

Delete self[key], without raising a `KeyError` if *key* does not exist.

__iter__ ()

Implement iter(self).

classmethod entity (*entity*: `DXFEntity`, *name*=`'DefaultDict'`, *appid*=`'EZDXF'`) → `Iterator[XDataUserDict]`

Context manager to manage a XDATA dict *name* for a given DXF *entity*. Appends the user dict to the existing `XData` instance or creates new `XData` instance.

Parameters

- **entity** (`DXFEntity`) – target DXF entity for the XDATA
- **name** (`str`) – name of the user list
- **appid** (`str`) – application specific AppID

commit () → `None`

Store all changes to the underlying `XData` instance. This call is not required if using the `entity()` context manager.

Raises

- `DXFValueError` – invalid chars `"\n"` or `"\r"` in a string
- `DXFTypeError` – invalid data type

Custom XRecord

The `UserRecord` and `BinaryRecord` classes help to store custom data in DXF files in `XRecord` objects a simple and safe way. This way requires DXF version R2000 or later, for DXF version R12 the only way to store custom data

is *Extended Data (XDATA)*.

The `UserRecord` stores Python types and nested container types: `int`, `float`, `str`, `Vec2`, `Vec3`, `list` and `dict`.

Requirements for Python structures:

- The top level structure has to be a `list`.
- Strings has to have max. 2049 characters and can not contain line breaks "`\n`" or "`\r`".
- Dict keys have to be simple Python types: `int`, `float`, `str`.

DXF Tag layout for Python types and structures stored in the `XRecord` object:

Only for the `UserRecord` the first tag is (2, user record name).

Type	DXF Tag(s)
<code>str</code>	(1, value) string with less than 2050 chars and including no line breaks
<code>int</code>	(90, value) int 32-bit, restricted by the DXF standard not by Python!
<code>float</code>	(40, value) "C" double
<code>Vec2</code>	(10, x), (20, y)
<code>Vec3</code>	(10, x) (20, y) (30, z)
<code>list</code>	starts with (2, "[") and ends with (2, "]")
<code>dict</code>	starts with (2, "{") and ends with (2, "}")

The `BinaryRecord` stores arbitrary binary data as `BLOB`.

Storage size limits of XRECORD according the DXF reference:

“This object is similar in concept to XDATA but is not limited by size or order.”

For usage look at this [example](#) at github or go to the tutorial: [Storing Custom Data in DXF Files](#).

See also:

- Tutorial: [Storing Custom Data in DXF Files](#)
- Example at [github](#)
- `ezdxf.entities.XRecord`

UserRecord

```
class ezdxf.urecord.UserRecord
```

xrecord

The underlying `XRecord` instance

name

The name of the `UserRecord`, an arbitrary string with less than 2050 chars and including no line breaks.

data

The Python data. The top level structure has to be a list (`MutableSequence`). Inside this container the following Python types are supported: `str`, `int`, `float`, `Vec2`, `Vec3`, `list`, `dict`

Nested data structures are supported list or/and dict in list or dict. Dict keys have to be simple Python types: `int`, `float`, `str`.

__init__ (*xrecord*: *ezdxf.entities.dxfobj.XRecord* = *None*, *, *name*: *str* = 'UserRecord', *doc*: *Drawing* = *None*)
Setup a *UserRecord* with the given *name*.

The data is stored in the given *xrecord* object, or in a new created *XRecord* instance if *None*. If *doc* is not *None* the new xrecord is added to the OBJECTS section of the DXF document.

Changes of the content has to be committed at the end to be stored in the underlying *xrecord* object.

Parameters

- **xrecord** (*XRecord*) – underlying *XRecord* instance, if *None* a new one will be created
- **name** (*str*) – name of the user list
- **doc** (*Drawing*) – DXF document or *None*

__str__()

Return str(*self*).

commit () → *XRecord*

Store *data* in the underlying *XRecord* instance. This call is not required if using the class by the *with* statement.

Raises

- *DXFValueError* – invalid chars "\n" or "\r" in a string
- *DXFTypError* – invalid data type

BinaryRecord

class *ezdxf.urecord.BinaryRecord*

xrecord

The underlying *XRecord* instance

data

The binary data as bytes, bytearray or memoryview.

__init__ (*xrecord*: *ezdxf.entities.dxfobj.XRecord* = *None*, *, *doc*: *Drawing* = *None*)
Setup a *BinaryRecord*.

The data is stored in the given *xrecord* object, or in a new created *XRecord* instance if *None*. If *doc* is not *None* the new xrecord is added to the OBJECTS section of the DXF document.

Changes of the content has to be committed at the end to be stored in the underlying *xrecord* object.

Parameters

- **xrecord** (*XRecord*) – underlying *XRecord* instance, if *None* a new one will be created
- **doc** (*Drawing*) – DXF document or *None*

__str__() → *str*

Return str(*self*).

commit () → *XRecord*

Store binary *data* in the underlying *XRecord* instance. This call is not required if using the class by the *with* statement.

6.8.10 Tools

Functions

DXF Unicode Decoder

The DXF format uses a special form of unicode encoding: “\U+xxxx”.

To avoid a speed penalty such encoded characters are not decoded automatically by the regular loading function:func:`ezdxf.readfile`, only the `recover` module does the decoding automatically, because this loading mode is already slow.

This kind of encoding is most likely used only in older DXF versions, because since DXF R2007 the whole DXF file is encoded in utf8 and a special unicode encoding is not necessary.

The `ezdxf.has_dxf_unicode()` and `ezdxf.decode_dxf_unicode()` are new support functions to decode unicode characters “\U+xxxx” manually.

`ezdxf.has_dxf_unicode(s: str) → bool`
Detect if string *s* contains encoded DXF unicode characters “\U+xxxx”.

`ezdxf.decode_dxf_unicode(s: str) → str`
Decode DXF unicode characters “\U+xxxx” in string *s*.

Tools

Some handy tool functions used internally by eздxf.

`ezdxf.tools.juliandate(date: datetime.datetime) → float`
`ezdxf.tools.calendardate(juliandate: float) → datetime.datetime`
`ezdxf.tools.set_flag_state(flags: int, flag: int, state: bool = True) → int`
Set/clear binary *flag* in data *flags*.

Parameters

- **flags** – data value
- **flag** – flag to set/clear
- **state** – `True` for setting, `False` for clearing

`ezdxf.tools.guid() → str`
Returns a general unique ID, based on `uuid.uuid4()`.

This function creates a GUID for the header variables \$VERSIONGUID and \$FINGERPRINTGUID, which matches the AutoCAD pattern {XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}.

`ezdxf.tools.bytes_to_hexstr(data: bytes) → str`
Returns *data* bytes as plain hex string.

`ezdxf.tools.suppress_zeros(s: str, leading: bool = False, trailing: bool = True)`
Suppress trailing and/or leading 0 of string *s*.

Parameters

- **s** – data string
- **leading** – suppress leading 0
- **trailing** – suppress trailing 0

```
ezdxf.tools.normalize_text_angle(angle: float, fix_upside_down=True) → float
Normalizes text angle to the range from 0 to 360 degrees and fixes upside down text angles.
```

Parameters

- **angle** – text angle in degrees
- **fix_upside_down** – rotate upside down text angle about 180 degree

SAT Format “Encryption”

```
ezdxf.tools.crypt.encode(text_lines: Iterable[str]) → Iterable[str]
Encode the Standard ACIS Text (SAT) format by AutoCAD “encryption” algorithm.
```

```
ezdxf.tools.crypt.decode(text_lines: Iterable[str]) → Iterable[str]
Decode the Standard ACIS Text (SAT) format “encrypted” by AutoCAD.
```

GfxAttribs

New in version 0.18.

The `ezdxf.gfxattribs` module provides the `GfxAttribs` class to create valid attribute dictionaries for the most often used DXF attributes supported by all graphical DXF entities. The advantage of using this class is auto-completion support by IDEs and an instant validation of the attribute values.

```
import ezdxf
from ezdxf.gfxattribs import GfxAttribs

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

attribs = GfxAttribs(layer="MyLayer", color=ezdxf.colors.RED)
line = msp.add_line((0, 0), (1, 0), dxfattribs=attribs)
circle = msp.add_circle((0, 0), radius=1.0, dxfattribs=attribs)

# Update DXF attributes of existing entities:
attribs = GfxAttribs(layer="MyLayer2", color=ezdxf.colors.BLUE)

# Convert GfxAttribs() to dict(), but this method cannot reset
# attributes to the default values like setting layer to "0".
line.update_dxf_attribs(dict(attribs))

# Using GfxAttribs.asdict(default_values=True), can reset attributes to the
# default values like setting layer to "0", except for true_color and
# transparency, which do not have default values, their absence is the
# default value.
circle.update_dxf_attribs(attribs.asdict(default_values=True))

# Remove true_color and transparency by assigning None
attribs.transparency = None # reset to transparency by layer!
attribs.rgb = None
```

Validation features:

- **layer** - string which can not contain certain characters: <>/\":;?*=`
- **color** - *AutoCAD Color Index (ACI)* value as integer in the range from 0 to 257
- **rgb** - true color value as (red, green, blue) tuple, all channel values as integer values in the range from 0 to 255

- **linetype** - string which can not contain certain characters: <>/\":;?*=` , does not check if the linetype exists
- **linewidth** - integer value in the range from 0 to 211, see [Lineweights](#) for valid values
- **transparency** - float value in the range from 0.0 to 1.0 and -1.0 for transparency by block
- **ltscale** - float value > 0.0

```
class ezdxf.gfxattribs.GfxAttribs(*, layer: str = '0', color: int = 256, rgb: Optional[Tuple[int, int, int]] = None, linetype: str = 'ByLayer', linewidth: int = -1, transparency: Optional[float] = None, ltscale: float = 1.0)
```

Represents often used DXF attributes of graphical entities.

New in version 0.18.

Parameters

- **layer** (*str*) – layer name as string
- **color** (*int*) – *AutoCAD Color Index (ACI)* color value as integer
- **rgb** – RGB true color (red, green, blue) tuple, each channel value in the range from 0 to 255, None for not set
- **linetype** (*str*) – linetype name, does not check if the linetype exist!
- **linewidth** (*int*) – see [Lineweights](#) documentation for valid values
- **transparency** (*float*) – transparency value in the range from 0.0 to 1.0, where 0.0 is opaque and 1.0 if fully transparent, -1.0 for transparency by block, None for transparency by layer
- **ltscale** (*float*) – linetype scaling value > 0.0, default value is 1.0

Raises DXFValueError – invalid attribute value

__str__() → str

Return str(self).

__repr__() → str

Return repr(self).

__iter__() → Iterator[Tuple[str, Any]]

Returns iter(self).

asdict (*default_values=False*) → Dict[str, Any]

Returns the DXF attributes as dict, returns also the default values if argument *default_values* is True. The true_color and transparency attributes do not have default values, the absence of these attributes is the default value.

items (*default_values=False*) → List[Tuple[str, Any]]

Returns the DXF attributes as list of name, value pairs, returns also the default values if argument *default_values* is True. The true_color and transparency attributes do not have default values, the absence of these attributes is the default value.

classmethod load_from_header (*doc: Drawing*) → GfxAttribs

Load default DXF attributes from the HEADER section.

There is no default true color value and the default transparency is not stored in the HEADER section.

Loads following header variables:

- \$CLAYER - current layer name
- \$CECOLOR - current ACI color

- \$CELTYPE - current linetype name
- \$CELWEIGHT - current linewidth
- \$CELTSCALE - current linetype scaling factor

write_to_header(*doc: Drawing*) → None

Write DXF attributes as default values to the HEADER section.

Writes following header variables:

- \$CLAYER - current layer name, if a layer table entry exist in *doc*
- \$CECOLOR - current ACI color
- \$CELTYPE - current linetype name, if a linetype table entry exist in *doc*
- \$CELWEIGHT - current linewidth
- \$CELTSCALE - current linetype scaling factor

classmethod from_entity(*entity: DXFEntity*) → GfxAttribs

Get the graphical attributes of an *entity* as *GfxAttribs* object.

Text Tools

MTextEditor

class ezdxf.tools.text.MTextEditor(*text: str = ''*)

The *MTextEditor* is a helper class to build MTEXT content strings with support for inline codes to change color, font or paragraph properties. The result is always accessible by the *text* attribute or the magic *__str__()* function as `str(MTextEditor("text"))`.

All text building methods return *self* to implement a floating interface:

```
e = MTextEditor("This example ").color("red").append("switches color to red.")
mtext = msp.add_mtext(str(e))
```

The initial text height, color, text style and so on is determined by the DXF attributes of the *MText* entity.

Warning: The *MTextEditor* assembles just the inline code, which has to be parsed and rendered by the target CAD application, *ezdxf* has no influence to that result.

Keep inline formatting as simple as possible, don't test the limits of its capabilities, this will not work across different CAD applications and keep the formatting in a logic manner like, do not change paragraph properties in the middle of a paragraph.

There is no official documentation for the inline codes!

Parameters **text** – init value of the MTEXT content string.

text

The MTEXT content as a simple string.

append(*text: str*) → ezdxf.tools.text.MTextEditor

Append *text*.

__iadd__(*text: str*) → ezdxf.tools.text.MTextEditor

Append *text*:

```
e = MTextEditor("First paragraph.\P")
e += "Second paragraph.\P")
```

`__str__()` → str

Returns the MTEXT content attribute `text`.

`clear()`

Reset the content to an empty string.

`font(name: str, bold: bool = False, italic: bool = False)` → ezdxf.tools.text.MTextEditor

Set the text font by the font family name. Changing the font height should be done by the `height()` or the `scale_height()` method. The font family name is the name shown in font selection widgets in desktop applications: “Arial”, “Times New Roman”, “Comic Sans MS”. Switching the codepage is not supported.

Parameters

- `name` – font family name
- `bold` – flag
- `italic` – flag

`height(height: float)` → ezdxf.tools.text.MTextEditor

Set the absolute text height in drawing units.

`scale_height(factor: float)` → ezdxf.tools.text.MTextEditor

Scale the text height by a `factor`. This scaling will accumulate, which means starting at height 2.5 and scaling by 2 and again by 3 will set the text height to $2.5 \times 2 \times 3 = 15$. The current text height is not stored in the `MTextEditor`, you have to track the text height by yourself! The initial text height is stored in the `MText` entity as DXF attribute `char_height`.

`width_factor(factor: float)` → ezdxf.tools.text.MTextEditor

Set the absolute text width factor.

`char_tracking_factor(factor: float)` → ezdxf.tools.text.MTextEditor

Set the absolute character tracking factor.

`oblique(angle: int)` → ezdxf.tools.text.MTextEditor

Set the text oblique angle in degrees, vertical is 0, a value of 15 will lean the text 15 degree to the right.

`color(name: str)` → ezdxf.tools.text.MTextEditor

Set the text color by color name: “red”, “yellow”, “green”, “cyan”, “blue”, “magenta” or “white”.

`aci(aci: int)` → ezdxf.tools.text.MTextEditor

Set the text color by `AutoCAD Color Index (ACI)` in range [0, 256].

`rgb(rgb: Tuple[int, int, int])` → ezdxf.tools.text.MTextEditor

Set the text color as RGB value.

`underline(text: str)` → ezdxf.tools.text.MTextEditor

Append `text` with a line below the text.

`overline(text: str)` → ezdxf.tools.text.MTextEditor

Append `text` with a line above the text.

`strike_through(text: str)` → ezdxf.tools.text.MTextEditor

Append `text` with a line through the text.

`group(text: str)` → ezdxf.tools.text.MTextEditor

Group `text`, all properties changed inside a group are reverted at the end of the group. AutoCAD supports grouping up to 8 levels.

stack (*upr*: str, *lwr*: str, *t*: str = '^') → ezdxf.tools.text.MTextEditor

Append stacked text *upr* over *lwr*, argument *t* defines the kind of stacking, the space " " after the “^” will be added automatically to avoid caret decoding:

```
"^": vertical stacked without divider line, e.g. \SA^ B:  
A  
B  
  
"/": vertical stacked with divider line, e.g. \SX/Y:  
X  
-  
Y  
  
"#": diagonal stacked, with slanting divider line, e.g. \S1#4:  
1/4
```

paragraph (*props*: ezdxf.tools.text.ParagraphProperties) → ezdxf.tools.text.MTextEditor

Set paragraph properties by a *ParagraphProperties* object.

bullet_list (*indent*: float, *bullets*: Iterable[str], *content*: Iterable[str]) → ezdxf.tools.text.MTextEditor

Build bulleted lists by utilizing paragraph indentation and a tabulator stop. Any string can be used as bullet. Indentation is a multiple of the initial MTEXT char height (see also docs about *ParagraphProperties*), which means indentation in drawing units is MText.dxf.char_height x *indent*.

Useful UTF bullets:

- “bull” U+2022 = • (Alt Numpad 7)
- “circle” U+25CB = (Alt Numpad 9)

For numbered lists just use numbers as bullets:

```
MTextEditor.bullet_list(  
    indent=2,  
    bullets=["1.", "2."],  
    content=["first", "second"],  
)
```

Parameters

- **indent** – content indentation as multiple of the initial MTEXT char height
- **bullets** – iterable of bullet strings, e.g. ["—"] * 3, for 3 dashes as bullet strings
- **content** – iterable of list item strings, one string per list item, list items should not contain new line or new paragraph commands.

Constants stored in the *MTextEditor* class:

NEW_LINE	'\P'
NEW_PARAGRAPH	'\P'
NEW_COLUMN	'\N'
UNDERLINE_START	'\L'
UNDERLINE_STOP	'\l'
OVERSTRIKE_START	'\O'
OVERSTRIKE_STOP	'\o'
STRIKE_START	'\K'
STRIKE_STOP	'\k'
ALIGN_BOTTOM	'\A0;'
ALIGN_MIDDLE	'\A1;'
ALIGN_TOP	'\A2;'
NBSP	'\~'
TAB	'^I'

```
class ezdxf.tools.text.ParagraphProperties(indent=0, left=0, right=0, align=DEFAULT, tab_stops=[])
```

Stores all known MTEXT paragraph properties in a NamedTuple. Indentations and tab stops are multiples of the default text height MText.dxf.char_height. E.g. char_height is 0.25 and indent is 4, the real indentation is $4 \times 0.25 = 1$ drawing unit. The default tabulator stops are 4, 8, 12, ... if no tabulator stops are explicit defined.

Parameters

- **indent** (*float*) – left indentation of the first line, relative to **left**, which means an indent of 0 has always the same indentation as **left**
- **left** (*float*) – left indentation of the paragraph except for the first line
- **right** (*float*) – left indentation of the paragraph
- **align** – *MTextParagraphAlignment* enum
- **tab_stops** – tuple of tabulator stops, as float or as str, float values are left aligned tab stops, strings with prefix "c" are center aligned tab stops and strings with prefix "r" are right aligned tab stops

tostring() → str

Returns the MTEXT paragraph properties as MTEXT inline code e.g. "\pxi-2,12;".

```
class ezdxf.lldxf.const.MTextParagraphAlignment
```

```
DEFAULT
LEFT
RIGHT
CENTER
JUSTIFIED
DISTRIBUTED
```

Single Line Text

```
class ezdxf.tools.text.TextLine(text: str, font: ezdxf.tools.fonts.AbstractFont)
```

Helper class which represents a single line text entity (e.g. *Text*).

Parameters

- **text** – content string
- **font** – ezdxf font definition like `MonospaceFont` or `MatplotlibFont`

stretch (*alignment*: str, *p1*: Vec3, *p2*: Vec3) → None

Set stretch factors for FIT and ALIGNED alignments to fit the text between *p1* and *p2*, only the distance between these points is important. Other given *alignment* values are ignore.

font_measurements () → ezdxf.tools.fonts.FontMeasurements

Returns the scaled font measurements.

baseline_vertices (*insert*, *halign*, *valign*, *angle*, *scale*) → List[Vec3]

Returns the left and the right baseline vertex of the text line.

Parameters

- **insert** – insertion location
- **halign** – horizontal alignment left=0, center=1, right=2
- **valign** – vertical alignment baseline=0, bottom=1, middle=2, top=3
- **angle** – text rotation in radians
- **scale** – scale in x- and y-axis as 2-tuple of float

corner_vertices (*insert*, *halign*, *valign*, *angle*, *scale*, *oblique*) → List[Vec3]

Returns the corner vertices of the text line in the order bottom left, bottom right, top right, top left.

Parameters

- **insert** – insertion location
- **halign** – horizontal alignment left=0, center=1, right=2
- **valign** – vertical alignment baseline=0, bottom=1, middle=2, top=3
- **angle** – text rotation in radians
- **scale** – scale in x- and y-axis as 2-tuple of float
- **oblique** – shear angle (slanting) in x-direction in radians

static transform_2d (*vertices*, *insert*, *shift*, *rotation*, *scale*, *oblique*) → List[Vec3]

Transform any vertices from the text line located at the base location at (0, 0) and alignment LEFT.

Parameters

- **vertices** – iterable of vertices
- **insert** – insertion location
- **shift** – (shift-x, shift-y) as 2-tuple of float
- **rotation** – text rotation in radians
- **scale** – (scale-x, scale-y) as 2-tuple of float
- **oblique** – shear angle (slanting) in x-direction in radians

Functions

`ezdxf.tools.text.caret_decode(text: str) → str`

DXF stores some special characters using caret notation. This function decodes this notation to normalise the representation of special characters in the string.

see: https://en.wikipedia.org/wiki/Caret_notation

`ezdxf.tools.text.estimate_mtext_content_extents(content: str, font: ezdxf.tools.fonts.AbstractFont, column_width: float = 0.0, line_spacing_factor: float = 1.0) → Tuple[float, float]`

Estimate the width and height of the `MText` content string. The result is very inaccurate if inline codes are used or line wrapping at the column border is involved! Column breaks `\N` will be ignored.

This function uses the optional *Matplotlib* package if available.

Parameters

- **content** – the `MText` content string
- **font** – font abstraction based on `ezdxf.tools.fonts.AbstractFont`
- **column_width** – `MText.dxf.width` or 0.0 for an unrestricted column width
- **line_spacing_factor** – `MText.dxf.line_spacing_factor`

Returns `Tuple[width, height]`

`ezdxf.tools.text.estimate_mtext_extents(mtext: MText) → Tuple[float, float]`

Estimate the width and height of a single column `MText` entity.

This function is faster than the `mtext_size()` function, but the result is very inaccurate if inline codes are used or line wrapping at the column border is involved!

This function uses the optional *Matplotlib* package if available.

Returns `Tuple[width, height]`

`ezdxf.tools.text.fast_plain_mtext(text: str, split=False) → Union[List[str], str]`

Returns the plain MTEXT content as a single string or a list of strings if `split` is True. Replaces `\P` by `\n` and removes other controls chars and inline codes.

This function is more than 4x faster than `plain_mtext()`, but does not remove single letter inline commands with arguments without a terminating semicolon like this "`\C1;red text`".

Note: Well behaved CAD applications and libraries always create inline codes for commands with arguments with a terminating semicolon like this "`\C1;red text`"!

Parameters

- **text** – MTEXT content string
- **split** – split content at line endings `\P`

`ezdxf.tools.text.is_text_vertical_stacked(text: DXFEntity) → bool`

Returns True if the associated text `Textstyle` is vertical stacked.

`ezdxf.tools.text.is_upside_down_text_angle(angle: float, tol: float = 3.0) → bool`

Returns True if the given text *angle* in degrees causes an upside down text in the [WCS](#). The strict flip range is $90^\circ < \text{angle} < 270^\circ$, the tolerance angle *tol* extends this range to: $90+\text{tol} < \text{angle} < 270-\text{tol}$. The angle is normalized to [0, 360).

Parameters

- **angle** – text angle in degrees
- **tol** – tolerance range in which text flipping will be avoided

`ezdxf.tools.text.leading(cap_height: float, line_spacing: float = 1.0) → float`

Returns the distance from baseline to baseline.

Parameters

- **cap_height** – cap height of the line
- **line_spacing** – line spacing factor as percentage of 3-on-5 spacing

`ezdxf.tools.text.plain_mtext(text: str, split=False, tabsize: int = 4) → Union[List[str], str]`

Returns the plain MTEXT content as a single string or a list of strings if *split* is True. Replaces \P by \n and removes other controls chars and inline codes.

This function is much slower than [`fast_plain_mtext\(\)`](#), but removes all inline codes.

Parameters

- **text** – MTEXT content string
- **split** – split content at line endings \P
- **tabsize** – count of replacement spaces for tabulators ^I

`ezdxf.tools.text.plain_text(text: str) → str`

Returns the plain text for [Text](#), [Attrib](#) and [Attdef](#) content.

`ezdxf.tools.text.safe_string(s: Optional[str], max_len: int = 255) → str`

Returns a string with line breaks \n replaced by \P and the length limited to *max_len*.

`ezdxf.tools.text.text_wrap(text: str, box_width: Optional[float], get_text_width: Callable[[str], float]) → List[str]`

Wrap text at \n and given *box_width*. This tool was developed for usage with the MTEXT entity. This isn't the most straightforward word wrapping algorithm, but it aims to match the behavior of AutoCAD.

Parameters

- **text** – text to wrap, included \n are handled as manual line breaks
- **box_width** – wrapping length, None to just wrap at \n
- **get_text_width** – callable which returns the width of the given string

`ezdxf.tools.text.upright_text_angle(angle: float, tol: float = 3.0) → float`

Returns a readable (upright) text angle in the range *angle* $\leq 90+\text{tol}$ or *angle* $\geq 270-\text{tol}$. The angle is normalized to [0, 360).

Parameters

- **angle** – text angle in degrees
- **tol** – tolerance range in which text flipping will be avoided

Text Size Tools

```
class ezdxf.tools.text_size.TextSize
```

A frozen dataclass as return type for the `text_size()` function.

width

The text width in drawing units (float).

cap_height

The font cap-height in drawing units (float).

total_height

The font total-height = cap-height + descender-height in drawing units (float).

```
ezdxf.tools.text_size.text_size(text: ezdxf.entities.text.Text) → ezdxf.tools.text_size.TextSize
```

Returns the measured text width, the font cap-height and the font total-height for a `Text` entity. This function uses the optional *Matplotlib* package if available to measure the final rendering width and font-height for the `Text` entity as close as possible. This function does not measure the real char height! Without access to the *Matplotlib* package the `MonospaceFont` is used and the measurements are very inaccurate.

See the `text2path` add-on for more tools to work with the text path objects created by the *Matplotlib* package.

```
class ezdxf.tools.text_size.MTextSize
```

A frozen dataclass as return type for the `mtext_size()` function.

total_width

The total width in drawing units (float)

total_height

The total height in drawing units (float), same as `max(column_heights)`.

column_width

The width of a single column in drawing units (float)

gutter_width

The space between columns in drawing units (float)

column_heights

A tuple of columns heights (float) in drawing units. Contains at least one column height and the the column height is 0 for an empty column.

column_count

The count of columns (int).

```
ezdxf.tools.text_size.mtext_size(mtext: ezdxf.entities.mtext.MText, tool: Op-
```

```
tional[ezdxf.tools.text_size.MTextSizeDetector] = None)
```

```
→ ezdxf.tools.text_size.MTextSize
```

Returns the total-width, -height and columns information for a `MText` entity.

This function uses the optional *Matplotlib* package if available to do font measurements and the internal text layout engine to determine the final rendering size for the `MText` entity as close as possible. Without access to the *Matplotlib* package the `MonospaceFont` is used and the measurements are very inaccurate.

Attention: The required full layout calculation is slow!

The first call to this function with *Matplotlib* support is very slow, because *Matplotlib* lookup all available fonts on the system. To speedup the calculation and accepting inaccurate results you can disable the *Matplotlib* support manually:

```
ezdxf.option.use_matplotlib = False
```

```
ezdxf.tools.text_size.estimate_mtext_extents (mtext: MText) → Tuple[float, float]
```

Estimate the width and height of a single column `MText` entity.

This function is faster than the `mtext_size()` function, but the result is very inaccurate if inline codes are used or line wrapping at the column border is involved!

This function uses the optional *Matplotlib* package if available.

Returns Tuple[width, height]

Fonts

The module `ezdxf.tools.fonts` manages the internal usage of fonts and has no relation how the DXF formats manages text styles.

See also:

The `Textstyle` entity, the DXF way to define fonts.

The tools in this module provide abstractions to get font measurements with and without the optional Matplotlib package.

For a proper text rendering the font measurements are required. *Ezdx*f has a lean approach to package dependencies, therefore the rendering results without support from the optional Matplotlib package are not very good.

Font Classes

```
ezdxf.tools.fonts.make_font (ttf_path: str, cap_height: float, width_factor: float = 1.0) → ezdxf.tools.fonts.AbstractFont
```

Factory function to create a font abstraction.

Creates a `MatplotlibFont` if the Matplotlib font support is available and enabled or else a `MonospaceFont`.

Parameters

- `ttf_path` – raw font file name as stored in the `Textstyle` entity
- `cap_height` – desired cap height in drawing units.
- `width_factor` – horizontal text stretch factor

```
class ezdxf.tools.fonts.AbstractFont (measurements: ezdxf.tools.fonts.FontMeasurements)
```

The *Ezdx*f font abstraction.

measurement

The `FontMeasurements` data.

```
text_width (text: str) → float
```

```
space_width () → float
```

```
class ezdxf.tools.fonts.MonospaceFont (cap_height: float, width_factor: float = 1.0, base_line: float = 0, descender_factor: float = 0.333, x_height_factor: float = 0.666)
```

Defines a monospaced font without knowing the real font properties. Each letter has the same cap- and descender height and the same width. This font abstraction is used if no Matplotlib font support is available.

Use the `make_font()` factory function to create a font abstraction.

```
text_width (text: str) → float
```

Returns the text width in drawing units for the given `text` based on a simple monospaced font calculation.

space_width() → float

Returns the width of a “space” char.

class ezdxf.tools.fonts.**MatplotlibFont** (*ttf_path*: str, *cap_height*: float = 1.0, *width_factor*: float = 1.0)

This class provides proper font measurement support by using the optional Matplotlib font support.

Use the [make_font\(\)](#) factory function to create a font abstraction.

text_width(*text*: str) → float

Returns the text width in drawing units for the given *text* string. Text rendering and width calculation is done by the Matplotlib TextPath class.

space_width() → float

Returns the width of a “space” char.

Font Anatomy

- A Visual Guide to the Anatomy of Typography: <https://visme.co/blog/type-anatomy/>
- Anatomy of a Character: <https://www.fonts.com/content/learning/fontology/level-1/type-anatomy/anatomy>

Font Properties

The default way of DXF to store fonts in the *Textstyle* entity by using the raw TTF file name is not the way how most render backends select fonts.

The render backends and web technologies select the fonts by their properties. This list shows the Matplotlib properties:

family List of font names in decreasing order of priority. The items may include a generic font family name, either “serif”, “sans-serif”, “cursive”, “fantasy”, or “monospace”.

style “normal” (“regular”), “italic” or “oblique”

stretch A numeric value in the range 0-1000 or one of “ultra-condensed”, “extra-condensed”, “condensed”, “semi-condensed”, “normal”, “semi-expanded”, “expanded”, “extra-expanded” or “ultra-expanded”

weight A numeric value in the range 0-1000 or one of “ultralight”, “light”, “normal”, “regular”, “book”, “medium”, “roman”, “semibold”, “demibold”, “demi”, “bold”, “heavy”, “extra bold”, “black”.

This way the backend can choose a similar font if the original font is not available.

See also:

- Matplotlib: https://matplotlib.org/stable/api/font_manager_api.html
- PyQt: <https://doc.qt.io/archives/qtforpython-5.12/PySide2/QtGui/QFont.html>
- W3C: <https://www.w3.org/TR/2018/REC-css-fonts-3-20180920/>

class ezdxf.tools.fonts.**FontFace** (*ttf*, *family*, *style*, *stretch*, *weight*)

This is the equivalent to the Matplotlib FontProperties class.

ttf

Raw font file name as string, e.g. “arial.ttf”

family

Family name as string, the default value is “sans-serif”

style

Font style as string, the default value is “normal”

stretch

Font stretch as string, the default value is “normal”

weight

Font weight as string, the default value is “normal”

class `ezdxf.tools.fonts.FontMeasurements`

See [Font Anatomy](#) for more information.

baseline

cap_height

x_height

descender_height

scale (*factor: float = 1.0*) → `ezdxf.tools.fonts.FontMeasurements`

scale_from_baseline (*desired_cap_height: float*) → `ezdxf.tools.fonts.FontMeasurements`

shift (*distance: float = 0.0*) → `ezdxf.tools.fonts.FontMeasurements`

Font Caching

*Ezdx*f uses Matplotlib to manage fonts and caches the collected information. The default installation of *ezdx*f provides a basic set of font properties. It is possible to create your own font cache specific for your system: see [`ezdxf.options.font_cache_directory`](#)

The font cache is loaded automatically at startup, if not disabled by setting config variable `auto_load_fonts` in `[core]` section to `False`: see [Environment Variables](#)

`ezdxf.tools.fonts.get_font_face(ttf_path: str, map_shx=True)` → `ezdxf.tools.fonts.FontFace`

Get cached font face definition by TTF file name e.g. “Arial.ttf”.

This function translates a DXF font definition by the raw TTF font file name into a `FontFace` object. Fonts which are not available on the current system gets a default font face.

Parameters

- **ttf_path** – raw font file name as stored in the `Textstyle` entity
- **map_shx** – maps SHX font names to TTF replacement fonts, e.g. “TXT” -> “txt_____ttf”

`ezdxf.tools.fonts.get_entity_font_face(entity: DXFEntity, doc=None)` → `ezdxf.tools.fonts.FontFace`

Returns the `FontFace` defined by the associated text style. Returns the default font face if the *entity* does not have or support the DXF attribute “style”. Supports the extended font information stored in `Textstyle` table entries.

Pass a DXF document as argument *doc* to resolve text styles for virtual entities which are not assigned to a DXF document. The argument *doc* always overrides the DXF document to which the *entity* is assigned to.

`ezdxf.tools.fonts.get_font_measurements(ttf_path: str, map_shx=True)` → `ezdxf.tools.fonts.FontMeasurements`

Get cached font measurements by TTF file name e.g. “Arial.ttf”.

Parameters

- **ttf_path** – raw font file name as stored in the `Textstyle` entity
- **map_shx** – maps SHX font names to TTF replacement fonts, e.g. “TXT” -> “txt_____ttf”

`ezdxf.tools.fonts.build_system_font_cache(*, path=None, rebuild=True) → None`

Build system font cache and save it to directory *path* if given. Set *rebuild* to `False` to just add new fonts. Requires the Matplotlib package!

A rebuild has to be done only after a new ezdxf installation, or new fonts were added to your system (which you want to use), or an update of ezdxf if you don't use your own external font cache directory.

See also: `ezdxf.options.font_cache_directory`

`ezdxf.tools.fonts.load(path=None, reload=False)`

Load all caches from given *path* or from default location, defined by `ezdxf.options.font_cache_directory` or the default cache from the `ezdxf.tools` folder.

This function is called automatically at startup if not disabled by environment variable `EZDXF_AUTO_LOAD_FONTS`.

`ezdxf.tools.fonts.save(path=None)`

Save all caches to given *path* or to default location, defined by `options.font_cache_directory` or into the `ezdxf.tools` folder.

6.8.11 Global Options

Global Options Object

The global `ezdxf` options are stored in the object `ezdxf.options`.

Recommended usage of the global options object:

```
import ezdxf
value = ezdxf.options.attribute
```

The `options` object uses the Standard Python class `ConfigParser` to manage the configuration. Shortcut attributes like `test_files` are simple properties and most shortcuts are read only marked by (Read only), read and writeable attributes are marked by (Read/Write).

To change options, especially the read only attributes, you have to edit the config file with a text editor, or set options by the `set()` method and write the current configuration into a config file.

Config Files

New in version 0.16.5.

The default config files are loaded from the user home directory as “`~/.config/ezdxf/ezdxf.ini`”, and the current working directory as “`./ezdxf.ini`”. A custom config file can be specified by the environment variable `EZDXF_CONFIG_FILE`. Ezdxf follows the [XDG Base Directory specification](#) if the environment variable `XDG_CONFIG_HOME` is set.

The config file loading order:

1. user home directory: “`~/.config/ezdxf/ezdxf.ini`”
2. current working directory: “`./ezdxf.ini`”
3. config file specified by `EZDXF_CONFIG_FILE`

A configuration file that is loaded later does not replace the previously loaded ones, only the existing options in the newly loaded file are added to the configuration and can overwrite existing options.

Configuration files are regular INI files, managed by the standard Python `ConfigParser` class.

File Structure:

```
[core]
default_dimension_text_style = OpenSansCondensed-Light
test_files = D:\Source\dxftest
font_cache_directory =
load_proxy_graphics = true
store_proxy_graphics = true
log_unprocessed_tags = false
filter_invalid_xdata_group_codes = true
write_fixed_meta_data_for_testing = false
disable_c_ext = false

[browse-command]
text_editor = "C:\Program Files\Notepad++\notepad++.exe" "{filename}" -n{num}
```

Modify and Save Changes

This code shows how to get and set values of the underlying ConfigParser object, but use the shortcut attributes if available:

```
# Set options, value has to be a str, use "true"/"false" for boolean values
ezdxf.options.set(section, key, value)

# Get option as string
value = ezdxf.options.get(section, key, default="")

# Special getter for boolean, int and float
value = ezdxf.options.get_bool(section, key, default=False)
value = ezdxf.options.get_int(section, key, default=0)
value = ezdxf.options.get_float(section, key, default=0.0)
```

If you set options, they are not stored automatically in a config file, you have to write back the config file manually:

```
# write back the default user config file "ezdxf.ini" in the
# user home directory
ezdxf.options.write_home_config()

# write back to the default config file "ezdxf.ini" in the
# current working directory
ezdxf.options.write_file()

# write back to a specific config file
ezdxf.options.write_file("my_config.ini")
# which has to be loaded manually at startup
ezdxf.options.read_file("my_config.ini")
```

This example shows how to change the `test_files` path and save the changes into a custom config file "my_config.ini":

```
import eздxf

test_files = Path("~/my-dxf-test-files").expand_user()
ezdxf.options.set(
    eздxf.options.CORE,  # section
    "test_files",  # key
```

(continues on next page)

(continued from previous page)

```

    "~/my-dxf-test-files", # value
)
ezdxf.options.write_file("my_config.ini")

```

Use a Custom Config File

You can specify a config file by the environment variable EZDXF_CONFIG_FILE, which is loaded after the default config files.

```
C:\> set EZDXF_CONFIG_FILE=D:\user\path\custom.ini
```

Custom config files are not loaded automatically like the default config files.

This example shows how to load the previous created custom config file “my_config.ini” from the current working directory:

```

import eздxf

ezdxf.options.read("my_config.ini")

```

That is all and because this is the last loaded config file, it overrides all default config files and the config file specified by EZDXF_CONFIG_FILE.

Functions

`ezdxf.options.set(section: str, key: str, value: str)`

Set option `key` in `section` to `values` as `str`.

`ezdxf.options.get(section: str, key: str, default: str = "") → str`

Get option `key` in `section` as string.

`ezdxf.options.get_bool(section: str, key: str, default: bool = False) → bool`

Get option `key` in `section` as bool.

`ezdxf.options.get_int(section: str, key: str, default: int = 0) → int`

Get option `key` in `section` as int.

`ezdxf.options.get_float(section: str, key: str, default: float = 0.0) → float`

Get option `key` in `section` as float.

`ezdxf.options.write(fp: TextIO)`

Write configuration into given file object `fp`, the file object must be a writeable text file with “utf8” encoding.

`ezdxf.options.write_file(filename: str = "ezdxf.ini")`

Write current configuration into file `filename`, default is “ezdxf.ini” in the current working directory.

`ezdxf.options.write_home_config()`

Write configuration into file “`~/.config/ezdxf/ezdxf.ini`”, `$XDG_CONFIG_HOME` is supported if set.

`ezdxf.options.read_file(filename: str)`

Append content from config file `filename`, but does not reset the configuration.

`ezdxf.options.print()`

Print configuration to `stdout`.

`ezdxf.options.reset()`

Reset options to factory default values.

```
ezdxf.options.delete_default_config_files()  
Delete the default config files "ezdxf.ini" in the current working and in the user home directory "~/config/ezdxf", $XDG_CONFIG_HOME is supported if set.  
  
ezdxf.options.preserve_proxy_graphics(state=True)  
Enable/disable proxy graphic load/store support by setting the options load_proxy_graphics and store_proxy_graphics to state.  
  
ezdxf.options.loaded_config_files  
Read only property of loaded config files as tuple for Path objects.
```

Core Options

For all core options the section name is `core`.

Default Dimension Text Style

The default dimension text style is used by the DIMENSION renderer of *ezdxf*, if the specified text style exist in the STYLE table. To use any of the default style of *ezdxf* you have to setup the styles at the creation of the DXF document: `ezdxf.new(setup=True)`, or setup the *ezdxf* default styles for a loaded DXF document:

```
import eздxf  
from eздxf.tool.standard import setup_drawing  
  
doc = eздxf.readfile("your.dxf")  
setup_drawing(doc)
```

Config file key: `default_dimension_text_style`

Shortcut attribute:

```
ezdxf.options.default_dimension_text_style  
(Read/Write) Get/Set default text style for DIMENSION rendering, default value is  
OpenSansCondensed-Light.
```

Font Cache Directory

*Ezdx*f has a bundled font cache to have faster access to font metrics. This font cache includes only fonts installed on the developing workstation. To add the fonts of your computer to this cache, you have to create your own external font cache. This has to be done only once after *ezdxf* was installed, or to add new installed fonts to the cache, and this requires the *Matplotlib* package.

This example shows, how to create an external font cache in the recommended directory of the XDG Base Directory specification: "`~/.cache/ezdxf`".

```
import eздxf  
from eздxf.tools import fonts  
  
# xdg_path() returns "$XDG_CACHE_HOME/ezdxf" or "~/.cache/ezdxf" if  
# $XDG_CACHE_HOME is not set  
font_cache_dir = eздxf.options.xdg_path("XDG_CACHE_HOME", ".cache")  
fonts.build_system_font_cache(path=font_cache_dir)  
ezdxf.options.font_cache_directory = font_cache_dir  
# Save changes to the default config file "~/.config/ezdxf/ezdxf.ini"
```

(continues on next page)

(continued from previous page)

```
# to load the font cache always from the new location.
ezdxf.options.write_home_config()
```

Config file key: `font_cache_directory`

Shortcut attribute:

`ezdxf.options.font_cache_directory`

(Read/Write) Get/set the font cache directory, if the directory is an empty string, the bundled font cache is used.
Expands “~” construct automatically.

Load Proxy Graphic

Proxy graphics are not essential for DXF files, but they can provide a simple graphical representation for complex entities, but extra memory is needed to store this information. You can save some memory by not loading the proxy graphic, but the proxy graphic is lost if you write back the DXF file.

The current version of *ezdxf* uses this proxy graphic to render MLEADER entities by the *drawing* add-on.

Config file key: `load_proxy_graphics`

Shortcut attribute:

`ezdxf.options.load_proxy_graphics`

(Read/Write) Load proxy graphics if True, default is True.

Store Proxy Graphic

Prevent exporting proxy graphics if set to False.

Config file key: `store_proxy_graphics`

Shortcut attribute:

`ezdxf.options.store_proxy_graphics`

(Read/Write) Export proxy graphics if True, default is True.

Debugging Options

For all debugging options the section name is `core`.

Test Files

Path to test files. Some of the **CADKit** test files are used by the integration tests, these files should be located in the `ezdxf.options.test_files_path` / "CADKitSamples" folder.

Config file key: `test_files`

Shortcut attributes:

`ezdxf.options.test_files`

(Read only) Returns the path to the *ezdxf* test files as str, expands “~” construct automatically.

`ezdxf.options.test_files_path`

(Read only) Path to test files as `pathlib.Path` object.

Filter Invalid XDATA Group Codes

Only a very limited set of group codes is valid in the XDATA section and AutoCAD is very picky about that. *Ezdx*f removes invalid XDATA group codes if this option is set to True, but this needs processing time, which is wasted if you get your DXF files from trusted sources like AutoCAD or BricsCAD.

Config file key: `filter_invalid_xdata_group_codes`

`ezdxf.options.filter_invalid_xdata_group_codes`

(Read only) Filter invalid XDATA group codes, default value is True.

Log Unprocessed Tags

Logs unprocessed DXF tags, this helps to find new and undocumented DXF features.

Config file key: `log_unprocessed_tags`

`ezdxf.options.log_unprocessed_tags`

(Read/Write) Log unprocessed DXF tags for debugging, default value is False.

Write Fixed Meta Data for Testing

Write the DXF files with fixed meta data to test your DXF files by a diff-like command, this is necessary to get always the same meta data like the saving time stored in the HEADER section. This may not work across different *ezdxf* versions!

Config file key: `write_fixed_meta_data_for_testing`

`ezdxf.options.write_fixed_meta_data_for_testing`

(Read/Write) Enable this option to always create same meta data for testing scenarios, e.g. to use a diff-like tool to compare DXF documents, default is False.

Disable C-Extension

It is possible to deactivate the optional C-extensions if there are any issues with the C-extensions. This has to be done in a default config file or by environment variable before the first import of *ezdxf*. For pypy3 the C-extensions are always disabled, because the JIT complied Python code is much faster.

Important: This option works only in the **default config files**, user config files which are loaded by `ezdxf.options.read_file()` cannot disable the C-Extensions, because at this point the setup process of *ezdxf* is already finished!

Config file key: `disable_c_ext`

`ezdxf.options.disable_c_ext`

(Read only) This option disables the C-extensions if True. This can only be done before the first import of *ezdxf* by using a config file or the environment variable `EZDXF_DISABLE_C_EXT`.

Use C-Extensions

`ezdxf.options.use_c_ext`

(Read only) Shows the actual state of C-extensions usage.

Use Matplotlib

This option can deactivate Matplotlib support for testing. This option is not stored in the `ConfigParser` object and is therefore not supported by config files!

Only attribute access is supported:

`ezdxf.options.use_matplotlib`

(Read/Write) Activate/deactivate Matplotlib support (e.g. for testing) if Matplotlib is installed, else `use_matplotlib` is always False.

Environment Variables

Some feature can be controlled by environment variables. Command line example for disabling the optional C-extensions on Windows:

```
C:\> set EZDXF_DISABLE_C_EXT=1
```

Important: If you change any environment variable, you have to restart the Python interpreter!

EZDXF_DISABLE_C_EXT Set environment variable `EZDXF_DISABLE_C_EXT` to 1 or True to disable the usage of the C-extensions.

EZDXF_TEST_FILES Path to the `ezdxf` test files required by some tests, for instance the `CADKit` sample files should be located in the `EZDXF_TEST_FILES/CADKitSamples` folder. See also option `ezdxf.options.test_files`.

EZDXF_CONFIG_FILE Specifies a user config file which will be loaded automatically after the default config files at the first import of `ezdxf`.

6.8.12 Miscellaneous

Zoom Layouts

New in version 0.16.

These functions mimic the ZOOM commands in CAD applications.

Zooming means resetting the current viewport limits to new values. The coordinates for the functions `center()` and `window()` are drawing units for the model space and paper space units for paper space layouts. The modelspace units in `Drawing.units` are ignored.

The extents detection for the functions `entities()` and `extents()` is done by the `ezdxf.bbox` module. Read the associated documentation to understand the limitations of the `ezdxf.bbox` module. Tl;dr The extents detection is **slow** and **not accurate**.

Because the ZOOM operations in CAD applications are not that precise, then zoom functions of this module uses the fast bounding box calculation mode of the `bbox` module, which means the argument `flatten` is always `False` for `extents()` function calls.

The region displayed by CAD applications also depends on the aspect ratio of the application window, which is not available to `ezdxf`, therefore the viewport size is just an educated guess of an aspect ratio of 2:1 (16:9 minus top toolbars and the bottom status bar).

Warning: All zoom functions replace the current viewport configuration by a single window configuration.

Example to reset the main CAD viewport of the model space to the extents of its entities:

```
import eздxf
from eздxf import zoom

doc = eздxf.new()
msp = doc.modelspace()
... # add your DXF entities

zoom.extents(msp)
doc.saveas("your.dxf")
```

`ezdxf.zoom.center` (*layout: Layout, point: Vertex, height: float*)

Resets the active viewport center of *layout* to the given *point*, argument *size* defines the width and height of the viewport. Replaces the current viewport configuration by a single window configuration.

`ezdxf.zoom.objects` (*layout: Layout, entities: Iterable[DXFEntity], factor: float=1*)

Resets the active viewport limits of *layout* to the extents of the given *entities*. Only entities in the given *layout* are taken into account. The argument *factor* scales the viewport limits. Replaces the current viewport configuration by a single window configuration.

`ezdxf.zoom.extents` (*layout: Layout, factor: float=1*)

Resets the active viewport limits of *layout* to the extents of all entities in this *layout*. The argument *factor* scales the viewport limits. Replaces the current viewport configuration by a single window configuration.

`ezdxf.zoom.window` (*layout: Layout, p1: Vertex, p2: Vertex*)

Resets the active viewport limits of *layout* to the lower left corner *p1* and the upper right corner *p2*. Replaces the current viewport configuration by a single window configuration.

Load DXF Comments

`ezdxf.comments.from_stream` (*stream: TextIO, codes: Set[int] = None*) → Iterable[DXFTag]

Yields comment tags from text *stream* as `DXFTag` objects.

Parameters

- **stream** – input text stream
- **codes** – set of group codes to yield additional DXF tags e.g. {5, 0} to also yield handle and structure tags

`ezdxf.comments.from_file` (*filename: str, codes: Set[int] = None*) → Iterable[DXFTag]

Yields comment tags from file *filename* as `DXFTag` objects.

Parameters

- **filename** – filename as string
- **codes** – yields also additional tags with specified group codes e.g. {5, 0} to also yield handle and structure tags

Reorder

Tools to reorder DXF entities by handle or a special sort handle mapping.

Such reorder mappings are stored only in layouts as `Modelspace`, `Paperspace` or `BlockLayout`, and can be retrieved by the method `get_redraw_order()`.

Each entry in the handle mapping replaces the actual entity handle, where the “0” handle has a special meaning, this handle always shows up at last in ascending ordering.

`ezdxf.reorder.ascendin(entities: Iterable[DXFGraphic], mapping: Union[Dict[KT, VT], Iterable[Tuple[str, str]]] = None) → Iterable[DXFGraphic]`

Yields entities in ascending handle order.

The sort handle doesn’t have to be the entity handle, every entity handle in `mapping` will be replaced by the given sort handle, `mapping` is an iterable of 2-tuples (entity_handle, sort_handle) or a dict (entity_handle, sort_handle). Entities with equal sort handles show up in source entities order.

Parameters

- **entities** – iterable of DXFGraphic objects
- **mapping** – iterable of 2-tuples (entity_handle, sort_handle) or a handle mapping as dict.

`ezdxf.reorder.descendin(entities: Iterable[DXFGraphic], mapping: Union[Dict[KT, VT], Iterable[Tuple[str, str]]] = None) → Iterable[DXFGraphic]`

Yields entities in descending handle order.

The sort handle doesn’t have to be the entity handle, every entity handle in `mapping` will be replaced by the given sort handle, `mapping` is an iterable of 2-tuples (entity_handle, sort_handle) or a dict (entity_handle, sort_handle). Entities with equal sort handles show up in reversed source entities order.

Parameters

- **entities** – iterable of DXFGraphic objects
- **mapping** – iterable of 2-tuples (entity_handle, sort_handle) or a handle mapping as dict.

6.9 Launcher

The command line script `ezdxf` launches various sub-commands:

pp	DXF pretty printer, replacement for the previous <code>dxfpp</code> command
audit	Audit and repair DXF files
draw	Draw and convert DXF files by the Matplotlib backend
view	PyQt DXF file viewer
browse	PyQt DXF structure browser for DXF debugging and curious people
strip	Strip comments and THUMBNAILIMAGE section from DXF files
config	Manage config files
info	Show information and optional stats of DXF files as loaded by eздxf

The help option `-h` is supported by the main script and all sub-commands:

```
C:\> eздxf -h
usage: eздxf [-h] [-V] [-v] [--config CONFIG] [--log LOG]
              {pp,audit,draw:view,browse,strip,config} ...

Command launcher for the Python package "eздxf":
https://pypi.org/project/eздxf/

positional arguments:
```

(continues on next page)

(continued from previous page)

```
{pp,audit,draw,view,browse,strip}
    pp                  pretty print DXF files as HTML file
    audit              audit and repair DXF files
    draw               draw and convert DXF files by Matplotlib
    view               view DXF files by the PyQt viewer
    browse             browse DXF file structure
    strip              strip comments from DXF files
    config             manage config files
    info               show information and optional stats of DXF files loaded by \_
ezdxf,
    this may not represent the original content of the file, use \_
the
    browse command to see the original content

optional arguments:
  -h, --help            show this help message and exit
  -V, --version         show version and exit
  -v, --verbose         give more output
  --config CONFIG       path to a config file
  --log LOG             path to a verbose appending log
```

Note: The `ezdxf` script is the only executable script installed on the user system, if installed by pip, the `dxfpp` script is not included anymore.

6.9.1 Pretty Printer

Pretty print the DXF text content as HTML file and open the file in the default web browser:

```
C:\> eздxf pp -o gear.dxf
```

[DXF Reference](#) | [Autodesk®](#) | [DWG TrueView](#) | [AutoCAD® Release History](#)

gear.dxf

[HEADER](#) | [CLASSES](#) | [TABLES](#) | [BLOCKS](#) | [ENTITIES](#) | [OBJECTS](#)

SECTION: HEADER

[previous](#) | [next](#) | [top](#)

```

9 <str> $ACADVER
1 <str> AC1027
9 <str> $ACADMAINTVER
70 <int> 105, b01101001
9 <str> $DWGCODEPAGE
3 <str> ANSI_1252
9 <str> $LASTSAVEDBY
1 <str> eздxf
9 <str> $REQUIREDVERSIONS
160 <int> 0
9 <str> $INSBASE
10 <point> (0.0, 0.0, 0.0)
9 <str> $EXTMIN
10 <point> (1e+20, 1e+20, 1e+20)
9 <str> $EXTMAX
10 <point> (-1e+20, -1e+20, -1e+20)
9 <str> $LIMMIN
10 <point> (0.0, 0.0)
9 <str> $LIMMAX
10 <point> (420.0, 297.0)

```

Print help:

```
C:\> eздxf pp -h
usage: eздxf pp [-h] [-o] [-r] [-x] [-l] [-s SECTION] FILE [FILE ...]

positional arguments:
  FILE                  DXF files pretty print

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

(continues on next page)

(continued from previous page)

```
-s SECTIONS, --sections SECTIONS
    choose sections to include and their order, h=HEADER, u
c=CLASSES,
t=TABLES, b=BLOCKS, e=ENTITIES, o=OBJECTS
```

6.9.2 Auditor

Audit and recover the DXF file “gear.dxf” and save the recovered version as “gear.rec.dxf”:

```
C:\> eздxf audit -s gear.dxf
auditing file: gear.dxf
No errors found.
Saved recovered file as: gear.rec.dxf
```

Print help:

```
C:\> eздxf audit -h
usage: eздxf audit [-h] [-s] FILE [FILE ...]

positional arguments:
  FILE      audit DXF files

optional arguments:
  -h, --help  show this help message and exit
  -s, --save   save recovered files with extension ".rec.dxf"
```

6.9.3 Draw

Convert the DXF file “gear.dxf” into a SVG file by the *Matplotlib* backend:

```
C:\> eздxf draw -o gear.svg gear.dxf
```

The “gear.svg” created by the *Matplotlib* backend:

Show all output formats supported by the *Matplotlib* backend on your system. This output may vary:

```
C:\> eздxf draw --formats
eps: Encapsulated Postscript
jpg: Joint Photographic Experts Group
jpeg: Joint Photographic Experts Group
pdf: Portable Document Format
pgf: PGF code for LaTeX
png: Portable Network Graphics
ps: Postscript
raw: Raw RGBA bitmap
rgba: Raw RGBA bitmap
svg: Scalable Vector Graphics
svgz: Scalable Vector Graphics
tif: Tagged Image File Format
tiff: Tagged Image File Format
```

Print help:

```
c:\> eздxf draw -h
usage: eздxf draw [-h] [--formats] [--layout LAYOUT] [--all-layers-visible]
                   [--all-entities-visible] [-o OUT] [--dpi DPI]
                   [--ltype {approximate,accurate}]
                   [FILE]

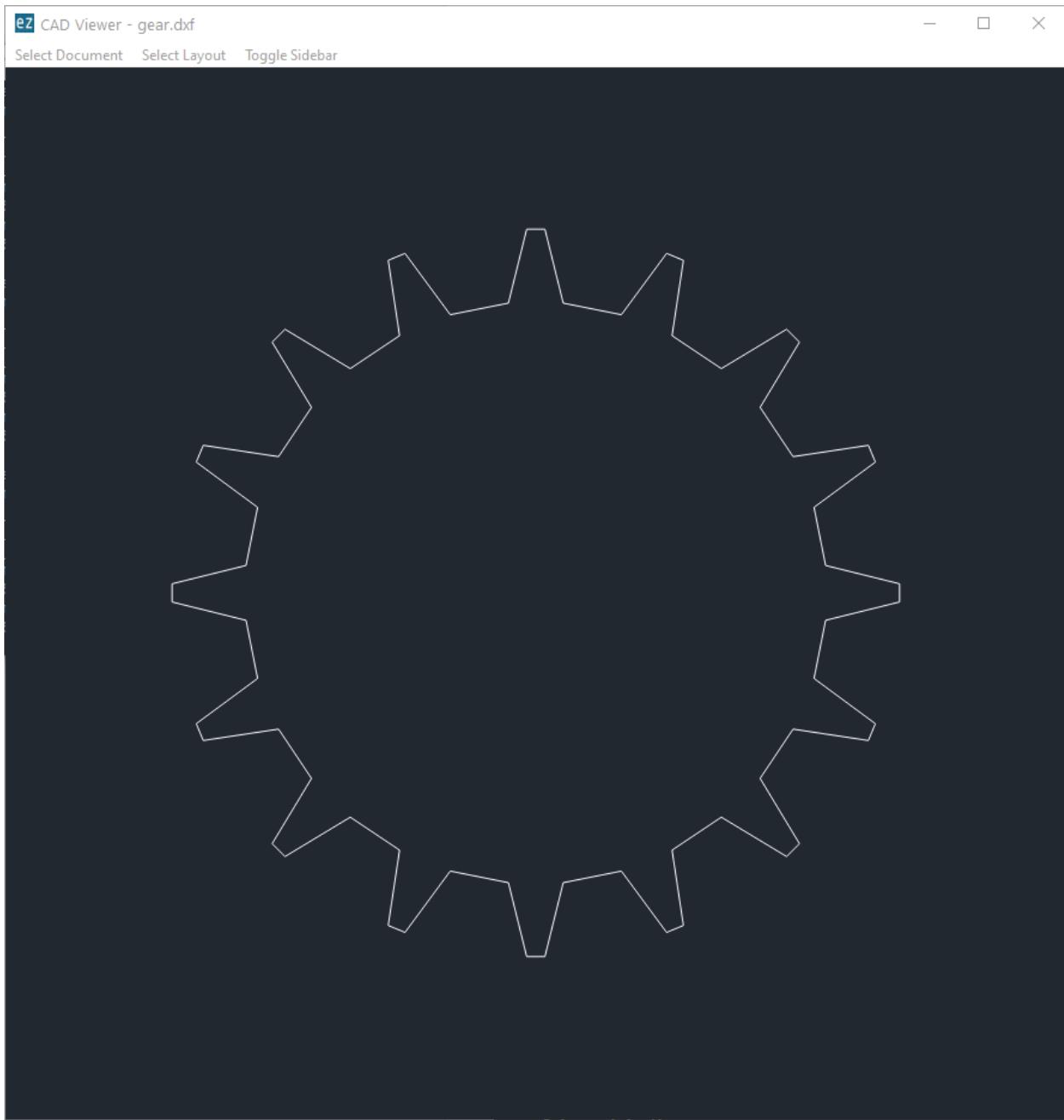
positional arguments:
  FILE                  DXF file to view or convert

optional arguments:
  -h, --help            show this help message and exit
  --formats             show all supported export formats and exit
  --layout LAYOUT       select the layout to draw
  --all-layers-visible draw all layers including the ones marked as invisible
  --all-entities-visible
                        draw all entities including the ones marked as
                        invisible (some entities are individually marked as
                        invisible even if the layer is visible)
  -o OUT, --out OUT    output filename for export
  --dpi DPI            target render resolution, default is 300
  --ltype {approximate,accurate}
                        select the line type rendering method, default is
                        approximate. Approximate uses the closest
                        approximation available to the backend, the accurate
                        method renders as accurately as possible but this
                        approach is slower.
```

6.9.4 View

View the DXF file “gear.dxf” by the *PyQt* backend:

```
C:\> eздxf view gear.dxf
```



Print help:

```
C:\> ezdxf view -h
usage: ezdxf view [-h] [--layout LAYOUT]
                  [--ltype {approximate,accurate}] [--lwscale LWSCALE]
                  [FILE]

positional arguments:
  FILE              DXF file to view

optional arguments:
  -h, --help         show this help message and exit
```

(continues on next page)

(continued from previous page)

```
--layout LAYOUT      select the layout to draw
--ltype {approximate,accurate}
                           select the line type rendering method, default
                           is approximate. Approximate uses the closest
                           approximation available to the backend, the
                           accurate method renders as accurately as
                           possible but this approach is slower.
--lwscale LWSCALE    set custom line weight scaling, default is 0 to
                           disable line weights at all
```

6.9.5 Browse

Browse the internal structure of a DXF file like a file system:

```
C:\> ezdxf browse gear.dxf
```

	Group Code	Data Type	Content
2029	0	<ctrl>	LWPOLYLINE
2031	5	<handle>	2F
2033	330	<ref>	17
2035	100	<ctrl>	AcDbEntity
2037	8	<str>	0
2039	100	<ctrl>	AcDbPolyline
2041	90	<int>	64
2043	70	<int>	1
2045	10	<point>	(9.99687255538428, -0.25007821057531726)
2049	10	<point>	(9.99687255538428, 0.25007821057531726)
2053	10	<point>	(7.964450219004663, 0.7533476680766843)
2057	10	<point>	(7.646486216427265, 2.3518606553084402)
2061	10	<point>	(9.331606731026024, 3.5945953621332394)
2065	10	<point>	(9.14020515506066, 4.0566796426884055)
2069	10	<point>	(7.069898873659621, 3.74386563811075)
2073	10	<point>	(6.164414002968977, 5.0990195135927845)
2077	10	<point>	(7.24568837309472, 6.892024376045111)
2081	10	<point>	(6.892024376045111, 7.24568837309472)
2085	10	<point>	(5.0990195135927845, 6.164414002968977)
2089	10	<point>	(3.7438656381107513, 7.069898873659621)

```
C:\> ezdxf browse -h
usage: ezdxf browse [-h] [-l LINE] [-g HANDLE] [FILE]

positional arguments:
  FILE                  DXF file to browse

optional arguments:
  -h, --help             show this help message and exit
```

(continues on next page)

(continued from previous page)

```
-l LINE, --line LINE  go to line number
-g HANDLE, --handle HANDLE
                    go to entity by HANDLE, HANDLE has to be a hex value without
                    any prefix like 'fefe'
```

The *browse* command stores options in the config file, e.g. for the *Notepad++* on Windows:

[browse-command]

```
text_editor = "C:\Program Files\Notepad++\notepad++.exe" "{filename}" -n{num}
icon_size = 32
```

`text_editor` is a simple format string: `text_editor.format(filename="test.dxf", num=100)`

Quote commands including spaces and always quote the filename argument!

For *gedit* on Linux use (untested):

[browse-command]

```
text_editor = gedit +{num} "{filename}"
icon_size = 32
```

The *browse* command opens a DXF structure browser to investigate the internals of a DXF file without interpreting the content. The functionality of the DXF browser is similar to the DXF *Pretty Printer* (*pp* command), but without the disadvantage of creating giant HTML files. The intended usage is debugging invalid DXF files, which can not be loaded by the `ezdxf.readfile()` or the `ezdxf.recover.readfile()` functions.

Line Numbers

The low level tag loader ignores DXF comments (group code 999). If there are comments in the DXF file the line numbers displayed in the DXF browser are not synchronized, use the `strip` command beforehand to remove all comments from the DXF file in order to keep the line numbers synchronized.

GUI Features

The tree view on the left shows the outline of the DXF file. The number in round brackets on the right side of each item shows the count of structure entities within the structure layer, the value in angle brackets on the left side is the entity handle.

The right list view shows the entity content as DXF tags. Structure tags (data type <ctrl>) are shown in blue, a double click on a reference handle (datatype <ref>) jumps to the referenced entity, reference handles of non-existent targets are shown in red.

Clicking on the first structure tag in the list opens the DXF reference provided by Autodesk in the standard web browser.

Auto Reload

The browser automatically displays a dialog for reloading DXF files if they have been modified by an external application.

Menus and Shortcuts

- **File Menu**
 - **Open DXF file...** *Ctrl+O*
 - **Reload DXF file** *Ctrl+R*
 - **Open in Text Editor** *Ctrl+T*, open the DXF file in the associated text editor at the current location
 - **Export DXF Entity...** *Ctrl+E*, export the current DXF entity shown in the list view as text file
 - **Copy DXF Entity to Clipboard** *Ctrl+C*, copy the current DXF entity shown in the list view into the clipboard
 - **Quit** *Ctrl+Q*
- **Navigate Menu**
 - **Go to Handle...** *Ctrl+G*
 - **Go to Line...** *Ctrl+L*
 - **Find Text...** *Ctrl+F*, opens the find text dialog
 - **Next Entity** *Ctrl+Right*, go to the next entity in the DXF structure
 - **Previous Entity** *Ctrl+Left*, go to the previous entity in the DXF structure
 - **Show Entity in TreeView** *Ctrl+Down*, expand the left tree view to the currently displayed entity in the list view - this does not happen automatically for performance reasons
 - **Entity History Back** *Alt+Left*
 - **Entity History Forward** *Alt+Right*
 - **Go to HEADERS Section** *Shift+H*
 - **Go to BLOCKS Section** *Shift+B*
 - **Go to ENTITIES Section** *Shift+E*
 - **Go to OBJECTS Section** *Shift+O*
- **Bookmarks Menu**
 - **Store Bookmark...** *Ctrl+Shift+B*, store current location as named bookmark
 - **Go to Bookmark...** *Ctrl+B*, go to stored location

6.9.6 Strip

Strip comment tags (group code 999) from ASCII DXF files and can remove the THUMBNAILIMAGE section. Binary DXF files are not supported.

```
C:\> ezdxf strip -h
usage: ezdxf strip [-h] [-b] [-v] FILE [FILE ...]

positional arguments:
  FILE          DXF file to process, wildcards "*" and "?" are supported

optional arguments:
  -h, --help      show this help message and exit
  -b, --backup    make a backup copy with extension ".bak" from the DXF file,
```

(continues on next page)

(continued from previous page)

```
overwrites existing backup files
-t, --thumbnail strip THUMBNAILIMAGE section
-v, --verbose give more output
```

6.9.7 Config

Manage config files.

```
C:\> eздxf config -h
usage: eздxf config [-h] [-p] [--home] [--reset]

optional arguments:
-h, --help    show this help message and exit
-p, --print   print configuration, to store the configuration use:
              "eздxf config -p > my.ini"
--home       create config file 'eздxf.ini' in the user home directory
              '~/.config/eздxf', $XDG_CONFIG_HOME is supported if set

--reset      factory reset, delete default config files 'eздxf.ini'
```

To create a new config file “my.ini” by printing the configuration into a file:

```
C:\> eздxf config -p > my.ini
```

6.9.8 Info

Show information and optional stats of DXF files as loaded by eздxf, this may not represent the original content of the file, use the [browse](#) command to see the original content. The upgrade is necessary for very old DXF versions prior to R12 and for the “special” versions R13 and R14. The *-s* option shows some statistics about the DXF content like entity count or table count. Use the *-v* option show more of everything.

```
C:\> eздxf info -h
usage: eздxf info [-h] [-v] [-s] FILE [FILE ...]

positional arguments:
FILE           DXF file to process, wildcards "*" and "?" are supported

options:
-h, --help      show this help message and exit
-v, --verbose   give more output
-s, --stats     show content stats
```

This is the verbose output for an old DXF R10 file and shows that the loading process created some required structures which do not exist in DXF R10 files, like the BLOCK_RECORD table or the OBJECTS section:

```
C:\> eздxf info -v -s test_R10.dxf

Filename: "test_R10.dxf"
Loaded content was upgraded from DXF Version AC1006 (R10)
Release: R12
DXF Version: AC1009
Maintenance Version: <undefined>
Codepage: ANSI_1252
```

(continues on next page)

(continued from previous page)

```

Encoding: cp1252
Unit system: Imperial
Modelspace units: Unitless
$LASTSAVEDBY: <undefined>
$HANDSEED: 0
$FINGERPRINTGUID: {9EADDC7C-5982-4C68-B770-8A62378C2B90}
$VERSIONGUID: {49336E63-D99B-45EC-803C-4D2BD03A7DE0}
$USERI1=0
$USERI2=0
$USERI3=0
$USERI4=0
$USERI5=0
$USERR1=0.0
$USERR2=0.0
$USERR3=0.0
$USERR4=0.0
$USERR5=0.0
File was not created by ezdxf >= 0.16.4
File was not written by ezdxf >= 0.16.4
Content stats:
LAYER table entries: 18
    0
    Defpoints
    LYR_00
    LYR_01
    LYR_02
    LYR_03
    LYR_04
    LYR_05
    LYR_06
    LYR_07
    LYR_08
    LYR_09
    LYR_10
    LYR_11
    LYR_12
    LYR_13
    LYR_14
    LYR_15
LTTYPE table entries: 13
    BORDER
    ByBlock
    ByLayer
    CENTER
    CONTINUOUS
    CUTTING
    DASHDOT
    DASHED
    DIVIDE
    DOT
    HIDDEN
    PHANTOM
    STITCH
STYLE table entries: 1
    STANDARD
DIMSTYLE table entries: 1
    Standard

```

(continues on next page)

(continued from previous page)

```
APPID table entries: 1
    ACAD
UCS table entries: 0
VIEW table entries: 0
VPORT table entries: 1
    *Active
BLOCK_RECORD table entries: 2
    *Model_Space
    *Paper_Space
Entities in modelspace: 78
    ARC (2)
    CIRCLE (2)
    LINE (74)
Entities in OBJECTS section: 20
    ACDBDICTIONARYWDFLT (1)
    ACDBPLACEHOLDER (1)
    DICTIONARY (11)
    LAYOUT (2)
    MATERIAL (3)
    MLEADERSTYLE (1)
    MLINESTYLE (1)
```

6.9.9 Show Version & Configuration

Show the *ezdxf* version and configuration:

```
C:\> eздxf -Vv

ezdxf v0.16.5b0 @ d:\source\ezdxf.git\src\ezdxf
Python version: 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26:21) [MSC v.1929 64 bit_
↪ (AMD64)]
using C-extensions: yes
using Matplotlib: yes

Configuration:
[core]
default_dimension_text_style = OpenSansCondensed-Light
test_files = D:\Source\dxftest
font_cache_directory =
load_proxy_graphics = true
store_proxy_graphics = true
log_unprocessed_tags = false
filter_invalid_xdata_group_codes = true
write_fixed_meta_data_for_testing = false
disable_c_ext = false

[browse-command]
text_editor = "C:\Program Files\Notepad++\notepad++.exe" "{filename}" -n{num}

Environment Variables:
EZDXF_DISABLE_C_EXT=
EZDXF_TEST_FILES=D:\Source\dxftest
EZDXF_CONFIG_FILE=
```

(continues on next page)

(continued from previous page)

Existing Configuration Files:
C:\Users\manfred\.config\ezdxf\ezdxf.ini

See also:

Documentation of the `ezdxf.options` module and the *Environment Variables*.

6.10 Rendering

The `ezdxf.render` subpackage provides helpful utilities to create complex forms.

- create complex meshes as `Mesh` entity.
- render complex curves like bezier curves, euler spirals or splines as `Polyline` entity
- vertex generators for simple and complex forms like circle, ellipse or euler spiral

Content

6.10.1 Spline

`class ezdxf.render.Spline(points: Iterable[Vertex] = None, segments: int = 100)`

This class can be used to render B-splines into DXF R12 files as approximated `Polyline` entities. The advantage of this class over the `R12Spline` class is, that this is a real 3D curve, which means that the B-spline vertices do have to be located in a flat plane, and no `UCS` class is needed to place the curve in 3D space.

See also:

The newer `Bspline` class provides the advanced vertex interpolation method `flattening()`.

`__init__(points: Iterable[Vertex] = None, segments: int = 100)`

Parameters

- **points** – spline definition points
- **segments** – count of line segments for approximation, vertex count is `segments + 1`

`subdivide(segments: int = 4) → None`

Calculate overall segment count, where `segments` is the sub-segment count, `segments = 4`, means 4 line segments between two definition points e.g. 4 definition points and 4 segments = 12 overall segments, useful for fit point rendering.

Parameters `segments` – sub-segments count between two definition points

`render_as_fit_points(layout: BaseLayout, degree: int = 3, method: str = 'chord', dxfattribs: dict = None) → None`

Render a B-spline as 2D/3D `Polyline`, where the definition points are fit points.

- 2D spline vertices uses: `add_polyline2d()`
- 3D spline vertices uses: `add_polyline3d()`

Parameters

- **layout** – `BaseLayout` object
- **degree** – degree of B-spline (`order = degree + 1`)

- **method** – “uniform”, “distance”/“chord”, “centripetal”/“sqrt_chord” or “arc” calculation method for parameter t
- **dxfattribs** – DXF attributes for *Polyline*

render_open_bspline (*layout*: *BaseLayout*, *degree*: *int* = 3, *dxfattribs*: *dict* = *None*) → *None*
Render an open uniform B-spline as 3D *Polyline*. Definition points are control points.

Parameters

- **layout** – *BaseLayout* object
- **degree** – degree of B-spline (order = *degree* + 1)
- **dxfattribs** – DXF attributes for *Polyline*

render_uniform_bspline (*layout*: *BaseLayout*, *degree*: *int* = 3, *dxfattribs*: *dict* = *None*) → *None*
Render a uniform B-spline as 3D *Polyline*. Definition points are control points.

Parameters

- **layout** – *BaseLayout* object
- **degree** – degree of B-spline (order = *degree* + 1)
- **dxfattribs** – DXF attributes for *Polyline*

render_closed_bspline (*layout*: *BaseLayout*, *degree*: *int* = 3, *dxfattribs*: *dict* = *None*) → *None*
Render a closed uniform B-spline as 3D *Polyline*. Definition points are control points.

Parameters

- **layout** – *BaseLayout* object
- **degree** – degree of B-spline (order = *degree* + 1)
- **dxfattribs** – DXF attributes for *Polyline*

render_open_rbspline (*layout*: *BaseLayout*, *weights*: *Iterable[float]*, *degree*: *int* = 3, *dxfattribs*: *dict* = *None*) → *None*
Render a rational open uniform BSpline as 3D *Polyline*. Definition points are control points.

Parameters

- **layout** – *BaseLayout* object
- **weights** – list of weights, requires a weight value (float) for each definition point.
- **degree** – degree of B-spline (order = *degree* + 1)
- **dxfattribs** – DXF attributes for *Polyline*

render_uniform_rbspline (*layout*: *BaseLayout*, *weights*: *Iterable[float]*, *degree*: *int* = 3, *dxfattribs*: *dict* = *None*) → *None*
Render a rational uniform B-spline as 3D *Polyline*. Definition points are control points.

Parameters

- **layout** – *BaseLayout* object
- **weights** – list of weights, requires a weight value (float) for each definition point.
- **degree** – degree of B-spline (order = *degree* + 1)
- **dxfattribs** – DXF attributes for *Polyline*

render_closed_rbspline (*layout*: *BaseLayout*, *weights*: *Iterable[float]*, *degree*: *int* = 3, *dxfattribs*: *dict* = *None*) → *None*
Render a rational B-spline as 3D *Polyline*. Definition points are control points.

Parameters

- **layout** – `BaseLayout` object
- **weights** – list of weights, requires a weight value (float) for each definition point.
- **degree** – degree of B-spline (order = $degree + 1$)
- **dxfattribs** – DXF attributes for `Polyline`

6.10.2 R12Spline

```
class ezdxf.render.R12Spline(control_points: Iterable[Vertex], degree: int = 2, closed: bool =
    True)
```

DXF R12 supports 2D B-splines, but Autodesk do not document the usage in the DXF Reference. The base entity for splines in DXF R12 is the POLYLINE entity. The spline itself is always in a plane, but as any 2D entity, the spline can be transformed into the 3D object by elevation and extrusion (OCS, UCS).

This way it was possible to store the spline parameters in the DXF R12 file, to allow CAD applications to modify the spline parameters and rerender the B-spline afterward again as polyline approximation. Therefore the result is not better than an approximation by the `Spline` class, it is also just a POLYLINE entity, but maybe someone need exact this tool in the future.

```
__init__(control_points: Iterable[Vertex], degree: int = 2, closed: bool = True)
```

Parameters

- **control_points** – B-spline control frame vertices
- **degree** – degree of B-spline, only 2 and 3 is supported
- **closed** – True for closed curve

```
render(layout: BaseLayout, segments: int = 40, ucs: UCS = None, dxfattribs: dict = None) → Polyline
```

Renders the B-spline into `layout` as 2D `Polyline` entity. Use an `UCS` to place the 2D spline in the 3D space, see `approximate()` for more information.

Parameters

- **layout** – `BaseLayout` object
- **segments** – count of line segments for approximation, vertex count is $segments + 1$
- **ucs** – `UCS` definition, control points in ucs coordinates.
- **dxfattribs** – DXF attributes for `Polyline`

```
approximate(segments: int = 40, ucs: UCS = None) → List[Vertex]
```

Approximate the B-spline by a polyline with `segments` line segments. If `ucs` is not `None`, `ucs` defines an `UCS`, to transformed the curve into `OCS`. The control points are placed xy-plane of the UCS, don't use z-axis coordinates, if so make sure all control points are in a plane parallel to the OCS base plane (UCS xy-plane), else the result is unpredictable and depends on the CAD application used to open the DXF file - it may crash.

Parameters

- **segments** – count of line segments for approximation, vertex count is $segments + 1$
- **ucs** – `UCS` definition, control points in ucs coordinates

Returns list of vertices in `OCS` as `Vec3` objects

6.10.3 Bezier

```
class ezdxf.render.Bezier
    Render a bezier curve as 2D/3D Polyline.
```

The `Bezier` class is implemented with multiple segments, each segment is an optimized 4 point bezier curve, the 4 control points of the curve are: the start point (1) and the end point (4), point (2) is start point + start vector and point (3) is end point + end vector. Each segment has its own approximation count.

See also:

The new `ezdxf.path` package provides many advanced construction tools based on the `Path` class.

start (*point: Vertex, tangent: Vertex*) → None
Set start point and start tangent.

Parameters

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

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

Parameters

- **point** – control point
- **tangent1** – first tangent as vector “left” of the control point
- **tangent2** – second tangent as vector “right” of the control point, if omitted *tangent2 = -tangent1*
- **segments** – count of line segments for the polyline approximation, count of line segments from the previous control point to the appended control point.

render (*layout: BaseLayout, force3d: bool = False, dxffattribs: dict = None*) → None
Render bezier curve as 2D/3D [Polyline](#).

Parameters

- **layout** – `BaseLayout` object
- **force3d** – force 3D polyline rendering
- **dxffattribs** – DXF attributes for [Polyline](#)

6.10.4 EulerSpiral

```
class ezdxf.render.EulerSpiral(curvature: float = 1)
    Render an euler spiral as a 3D Polyline or a Spline entity.
```

This is a parametric curve, which always starts at the origin (0, 0).

__init__ (*curvature: float = 1*)

Parameters **curvature** – Radius of curvature

render_polyline (*layout: BaseLayout, length: float = 1, segments: int = 100, matrix: Matrix44 = None, dxffattribs: dict = None*)
Render curve as [Polyline](#).

Parameters

- **layout** – `BaseLayout` object
- **length** – length measured along the spiral curve from its initial position
- **segments** – count of line segments to use, vertex count is $segments + 1$
- **matrix** – transformation matrix as `Matrix44`
- **dxfattribs** – DXF attributes for `Polyline`

Returns `Polyline`

```
ezdxf.render.spline(layout: BaseLayout, length: float = 1, fit_points: int = 10, degree: int = 3, matrix: Matrix44 = None, dxfattribs: dict = None)
    Render curve as Spline.
```

Parameters

- **layout** – `BaseLayout` object
- **length** – length measured along the spiral curve from its initial position
- **fit_points** – count of spline fit points to use
- **degree** – degree of B-spline
- **matrix** – transformation matrix as `Matrix44`
- **dxfattribs** – DXF attributes for `Spline`

Returns `Spline`

6.10.5 Random Paths

Random path generators for testing purpose.

```
ezdxf.render.random_2d_path(steps=100, max_step_size=1, max_heading=pi/2, retarget=20) →
    Iterable[Vec2]
```

Returns a random 2D path as iterable of `Vec2` objects.

Parameters

- **steps** – count of vertices to generate
- **max_step_size** – max step size
- **max_heading** – limit heading angle change per step to $\pm max_heading/2$ in radians
- **retarget** – specifies steps before changing global walking target

```
ezdxf.render.random_3d_path(steps=100, max_step_size=1, max_heading=pi/2, max_pitch=pi/8,
    retarget=20) → Iterable[Vec3]
```

Returns a random 3D path as iterable of `Vec3` objects.

Parameters

- **steps** – count of vertices to generate
- **max_step_size** – max step size
- **max_heading** – limit heading angle change per step to $\pm max_heading/2$, rotation about the z-axis in radians
- **max_pitch** – limit pitch angle change per step to $\pm max_pitch/2$, rotation about the x-axis in radians
- **retarget** – specifies steps before changing global walking target

6.10.6 Forms

This module provides functions to create 2D and 3D forms as vertices or mesh objects.

2D Forms

- `circle()`
- `square()`
- `box()`
- `ellipse()`
- `euler_spiral()`
- `ngon()`
- `star()`
- `gear()`

3D Forms

- `cube()`
- `cylinder()`
- `cylinder_2p()`
- `cone()`
- `cone_2p()`
- `sphere()`

3D Form Builder

- `extrude()`
- `from_profiles_linear()`
- `from_profiles_spline()`
- `rotation_form()`

2D Forms

Basic 2D shapes as iterable of `Vec3`.

```
ezdxf.render.forms.circle(count: int, radius: float = 1, elevation: float = 0, close: bool = False)
    → Iterable[Vec3]
```

Create polygon vertices for a `circle` with the given `radius` and approximated by `count` vertices, `elevation` is the z-axis for all vertices.

Parameters

- **count** – count of polygon vertices
- **radius** – circle radius
- **elevation** – z-axis for all vertices
- **close** – yields first vertex also as last vertex if True.

Returns vertices in counter clockwise orientation as `Vec3` objects

`ezdxf.render.forms.square(size: float = 1.) → Tuple[Vec3, Vec3, Vec3, Vec3]`

Returns 4 vertices for a square with a side length of the given `size`, lower left corner is $(0, 0)$, upper right corner is $(size, size)$.

`ezdxf.render.forms.box(sx: float = 1., sy: float = 1.) → Tuple[Vec3, Vec3, Vec3, Vec3]`

Returns 4 vertices for a box with a width of `sx` by and a height of `sy`, lower left corner is $(0, 0)$, upper right corner is (sx, sy) .

`ezdxf.render.forms.ellipse(count: int, rx: float = 1, ry: float = 1, start_param: float = 0, end_param: float = 2 * pi, elevation: float = 0) → Iterable[Vec3]`

Create polygon vertices for an `ellipse` with given `rx` as x-axis radius and `ry` as y-axis radius approximated by `count` vertices, `elevation` is the z-axis for all vertices. The ellipse goes from `start_param` to `end_param` in counter clockwise orientation.

Parameters

- `count` – count of polygon vertices
- `rx` – ellipse x-axis radius
- `ry` – ellipse y-axis radius
- `start_param` – start of ellipse in range $[0, 2\pi]$
- `end_param` – end of ellipse in range $[0, 2\pi]$
- `elevation` – z-axis for all vertices

Returns vertices in counter clockwise orientation as `Vec3` objects

`ezdxf.render.forms.euler_spiral(count: int, length: float = 1, curvature: float = 1, elevation: float = 0) → Iterable[Vec3]`

Create polygon vertices for an `euler spiral` of a given `length` and radius of curvature. This is a parametric curve, which always starts at the origin $(0, 0)$.

Parameters

- `count` – count of polygon vertices
- `length` – length of curve in drawing units
- `curvature` – radius of curvature
- `elevation` – z-axis for all vertices

Returns vertices as `Vec3` objects

`ezdxf.render.forms.ngon(count: int, length: float = None, radius: float = None, rotation: float = 0., elevation: float = 0., close: bool = False) → Iterable[Vec3]`

Returns the corner vertices of a `regular polygon`. The polygon size is determined by the edge `length` or the circum `radius` argument. If both are given `length` has the higher priority.

Parameters

- `count` – count of polygon corners ≥ 3
- `length` – length of polygon side
- `radius` – circum radius
- `rotation` – rotation angle in radians
- `elevation` – z-axis for all vertices
- `close` – yields first vertex also as last vertex if True.

Returns vertices as `Vec3` objects

```
ezdxf.render.forms.star(count: int, r1: float, r2: float, rotation: float = 0., elevation: float = 0.,
                        close: bool = False) → Iterable[Vec3]
```

Returns the corner vertices for a [star shape](#).

The shape has *count* spikes, *r1* defines the radius of the “outer” vertices and *r2* defines the radius of the “inner” vertices, but this does not mean that *r1* has to be greater than *r2*.

Parameters

- **count** – spike count ≥ 3
- **r1** – radius 1
- **r2** – radius 2
- **rotation** – rotation angle in radians
- **elevation** – z-axis for all vertices
- **close** – yields first vertex also as last vertex if True.

Returns vertices as [Vec3](#) objects

```
ezdxf.render.forms.gear(count: int, top_width: float, bottom_width: float, height: float, outside_radius: float, elevation: float = 0, close: bool = False) → Iterable[Vec3]
```

Returns the corner vertices of a [gear shape](#) (cogwheel).

Warning: This function does not create correct gears for mechanical engineering!

Parameters

- **count** – teeth count ≥ 3
- **top_width** – teeth width at outside radius
- **bottom_width** – teeth width at base radius
- **height** – teeth height; base radius = outside radius - height
- **outside_radius** – outside radius
- **elevation** – z-axis for all vertices
- **close** – yields first vertex also as last vertex if True.

Returns vertices in counter clockwise orientation as [Vec3](#) objects

3D Forms

Create 3D forms as [MeshTransformer](#) objects.

```
ezdxf.render.forms.cube(center: bool = True) → MeshTransformer
```

Create a [cube](#) as [MeshTransformer](#) object.

Parameters **center** – ‘mass’ center of cube, $(0, 0, 0)$ if True, else first corner at $(0, 0, 0)$

Returns: [MeshTransformer](#)

```
ezdxf.render.forms.cylinder(count: int, radius: float = 1., top_radius: float = None, top_center: Vertex = (0, 0, 1), caps=True, ngons=True) → MeshTransformer
```

Create a [cylinder](#) as [MeshTransformer](#) object, the base center is fixed in the origin $(0, 0, 0)$.

Parameters

- **count** – profiles edge count
- **radius** – radius for bottom profile
- **top_radius** – radius for top profile, if `None` `top_radius == radius`
- **top_center** – location vector for the center of the top profile
- **caps** – close hull with bottom cap and top cap (as N-gons)
- **ngons** – use ngons for caps if `True` else subdivide caps into triangles

Returns: `MeshTransformer`

`ezdxf.render.forms.cylinder_2p(count: int = 16, radius: float = 1, base_center=(0, 0, 0), top_center=(0, 0, 1))` → `MeshTransformer`

Create a `cylinder` as `MeshTransformer` object from two points, `base_center` is the center of the base circle and, `top_center` the center of the top circle.

Parameters

- **count** – profiles edge count
- **radius** – radius for bottom profile
- **base_center** – center of base circle
- **top_center** – center of top circle

Returns: `MeshTransformer`

`ezdxf.render.forms.cone(count: int, radius: float, apex: Vertex = (0, 0, 1), caps=True, ngons=True)` → `MeshTransformer`

Create a `cone` as `MeshTransformer` object, the base center is fixed in the origin (0, 0, 0).

Parameters

- **count** – edge count of basis_vector
- **radius** – radius of basis_vector
- **apex** – tip of the cone
- **caps** – add a bottom face if `True`
- **ngons** – use ngons for caps if `True` else subdivide caps into triangles

Returns: `MeshTransformer`

`ezdxf.render.forms.cone_2p(count: int, radius: float, apex: Vertex = (0, 0, 1))` → `MeshTransformer`

Create a `cone` as `MeshTransformer` object from two points, `base_center` is the center of the base circle and `apex` as the tip of the cone.

Parameters

- **count** – edge count of basis_vector
- **radius** – radius of basis_vector
- **base_center** – center point of base circle
- **apex** – tip of the cone

Returns: `MeshTransformer`

`ezdxf.render.forms.sphere(count: int = 16, stacks: int = 8, radius: float = 1, quads=True)` → `MeshTransformer`

Create a `sphere` as `MeshTransformer` object, center is fixed at origin (0, 0, 0).

Parameters

- **count** – longitudinal slices
- **stacks** – latitude slices
- **radius** – radius of sphere
- **quads** – use quads for body faces if True else triangles

Returns: *MeshTransformer*

3D Form Builder

`ezdxf.render.forms.extrude(profile: Iterable[Vertex], path: Iterable[Vertex], close=True) → MeshTransformer`
Extrude a *profile* polygon along a *path* polyline, vertices of profile should be in counter clockwise order.

Parameters

- **profile** – sweeping profile as list of (x, y, z) tuples in counter clockwise order
- **path** – extrusion path as list of (x, y, z) tuples
- **close** – close profile polygon if True

Returns: *MeshTransformer*

`ezdxf.render.forms.from_profiles_linear(profiles: Iterable[Iterable[Vertex]], close=True, caps=False, ngons=True) → MeshTransformer`
Create MESH entity by linear connected *profiles*.

Parameters

- **profiles** – list of profiles
- **close** – close profile polygon if True
- **caps** – close hull with bottom cap and top cap
- **ngons** – use ngons for caps if True else subdivide caps into triangles

Returns: *MeshTransformer*

`ezdxf.render.forms.from_profiles_spline(profiles: Iterable[Iterable[Vertex]], subdivide: int = 4, close=True, caps=False, ngons=True) → MeshTransformer`

Create MESH entity by spline interpolation between given *profiles*. Requires at least 4 profiles. A subdivide value of 4, means, create 4 face loops between two profiles, without interpolation two profiles create one face loop.

Parameters

- **profiles** – list of profiles
- **subdivide** – count of face loops
- **close** – close profile polygon if True
- **caps** – close hull with bottom cap and top cap
- **ngons** – use ngons for caps if True else subdivide caps into triangles

Returns: *MeshTransformer*

```
ezdxf.render.forms.rotation_form(count: int, profile: Iterable[Vertex], angle: float = 2 * pi, axis:
                                  Vertex = (1, 0, 0)) → MeshTransformer
Create MESH entity by rotating a profile around an axis.
```

Parameters

- **count** – count of rotated profiles
- **profile** – profile to rotate as list of vertices
- **angle** – rotation angle in radians
- **axis** – rotation axis

Returns: *MeshTransformer*

6.10.7 MeshBuilder

The *MeshBuilder* is a helper class to create *Mesh* entities. Stores a list of vertices and a faces list where each face is a list of indices into the vertices list.

The *MeshBuilder.render()* method, renders the mesh into a *Mesh* entity. The *Mesh* entity supports ngons in AutoCAD, ngons are polygons with more than 4 vertices.

The basic *MeshBuilder* class does not support transformations.

```
class ezdxf.render.MeshBuilder
```

vertices

List of vertices as *Vec3* or (x, y, z) tuple

faces

List of faces as list of vertex indices, where a vertex index is the index of the vertex in the *vertices* list.
A face requires at least three vertices, *Mesh* supports ngons, so the count of vertices is not limited.

copy()

Returns a copy of mesh.

faces_as_vertices() → Iterable[List[Vec3]]

Iterate over all mesh faces as list of vertices.

add_vertices(vertices: Iterable[Vertex]) → Sequence[int]

Add new vertices to the mesh, each vertex is a (x, y, z) tuple or a *Vec3* object, returns the indices of the *vertices* added to the *vertices* list.

e.g. adding 4 vertices to an empty mesh, returns the indices (0, 1, 2, 3), adding additional 4 vertices returns the indices (4, 5, 6, 7).

Parameters **vertices** – list of vertices, vertex as (x, y, z) tuple or *Vec3* objects

Returns indices of the *vertices* added to the *vertices* list

Return type tuple

add_face(vertices: Iterable[Vertex]) → None

Add a face as vertices list to the mesh. A face requires at least 3 vertices, each vertex is a (x, y, z) tuple or *Vec3* object. The new vertex indices are stored as face in the *faces* list.

Parameters **vertices** – list of at least 3 vertices [(x1, y1, z1), (x2, y2, z2),
(x3, y3, z3), ...]

add_mesh (*vertices=None, faces=None, edges=None, mesh=None*) → None

Add another mesh to this mesh.

A *mesh* can be a *MeshBuilder*, *MeshVertexMerger* or *Mesh* object or requires the attributes *vertices* and *faces*.

Parameters

- **vertices** – list of vertices, a vertex is a (x, y, z) tuple or *Vec3* object
- **faces** – list of faces, a face is a list of vertex indices
- **mesh** – another mesh entity

has_none_planar_faces () → bool

Returns True if any face is none planar.

render_mesh (*layout: BaseLayout, dxfattribs: dict = None, matrix: Matrix44 = None, ucs: UCS = None*)

Render mesh as *Mesh* entity into *layout*.

Parameters

- **layout** – *BaseLayout* object
- **dfattribs** – dict of DXF attributes e.g. {'layer': 'mesh', 'color': 7}
- **matrix** – transformation matrix of type *Matrix44*
- **ucs** – transform vertices by *UCS* to *WCS*

render_polyface (*layout: BaseLayout, dxfattribs: dict = None, matrix: Matrix44 = None, ucs: UCS = None*)

Render mesh as *Polyface* entity into *layout*.

Parameters

- **layout** – *BaseLayout* object
- **dfattribs** – dict of DXF attributes e.g. {'layer': 'mesh', 'color': 7}
- **matrix** – transformation matrix of type *Matrix44*
- **ucs** – transform vertices by *UCS* to *WCS*

render_3dfaces (*layout: BaseLayout, dxfattribs: dict = None, matrix: Matrix44 = None, ucs: UCS = None*)

Render mesh as *Face3d* entities into *layout*.

Parameters

- **layout** – *BaseLayout* object
- **dfattribs** – dict of DXF attributes e.g. {'layer': 'mesh', 'color': 7}
- **matrix** – transformation matrix of type *Matrix44*
- **ucs** – transform vertices by *UCS* to *WCS*

render_normals (*layout: BaseLayout, length: float = 1, relative=True, dxfattribs: dict = None*)

Render face normals as *Line* entities into *layout*, useful to check orientation of mesh faces.

Parameters

- **layout** – *BaseLayout* object

- **length** – visual length of normal, use length < 0 to point normals in opposite direction
- **relative** – scale length relative to face size if True
- **dxfattribs** – dict of DXF attributes e.g. {'layer': 'normals', 'color': 6}

classmethod from_mesh (other: Union[MeshBuilder, Mesh]) → T

Create new mesh from other mesh as class method.

Parameters **other** – mesh of type `MeshBuilder` and inherited or DXF `Mesh` entity or any object providing attributes `vertices`, `edges` and `faces`.

classmethod from_polyface (other: Union[Polymesh, Polyface]) → T

Create new mesh from a `Polyface` or `Polymesh` object.

classmethod from_builder (other: MeshBuilder) → T

Create new mesh from other mesh builder, faster than `from_mesh ()` but supports only `MeshBuilder` and inherited classes.

6.10.8 MeshTransformer

Same functionality as `MeshBuilder` but supports inplace transformation.

class ezdxf.render.MeshTransformer

Subclass of `MeshBuilder`

subdivide (level: int = 1, quads=True) → MeshTransformer

Returns a new `MeshTransformer` object with subdivided faces and edges.

Parameters

- **level** – subdivide levels from 1 to max of 5
- **quads** – create quad faces if True else create triangles

transform (matrix: Matrix44)

Transform mesh inplace by applying the transformation `matrix`.

Parameters **matrix** – 4x4 transformation matrix as `Matrix44` object

translate (dx: float = 0, dy: float = 0, dz: float = 0)

Translate mesh inplace.

Parameters

- **dx** – translation in x-axis
- **dy** – translation in y-axis
- **dz** – translation in z-axis

scale (sx: float = 1, sy: float = 1, sz: float = 1)

Scale mesh inplace.

Parameters

- **sx** – scale factor for x-axis
- **sy** – scale factor for y-axis
- **sz** – scale factor for z-axis

scale_uniform (s: float)

Scale mesh uniform inplace.

Parameters **s** – scale factor for x-, y- and z-axis

rotate_x (*angle: float*)

Rotate mesh around x-axis about *angle* inplace.

Parameters **angle** – rotation angle in radians

rotate_y (*angle: float*)

Rotate mesh around y-axis about *angle* inplace.

Parameters **angle** – rotation angle in radians

rotate_z (*angle: float*)

Rotate mesh around z-axis about *angle* inplace.

Parameters **angle** – rotation angle in radians

rotate_axis (*axis: Vertex, angle: float*)

Rotate mesh around an arbitrary axis located in the origin (0, 0, 0) about *angle*.

Parameters

- **axis** – rotation axis as Vec3

- **angle** – rotation angle in radians

6.10.9 MeshVertexMerger

Same functionality as [MeshBuilder](#), but created meshes with unique vertices and no doublets, but [MeshVertexMerger](#) needs extra memory for bookkeeping and also does not support transformations. Location of merged vertices is the location of the first vertex with the same key.

This class is intended as intermediate object to create a compact meshes and convert them to [MeshTransformer](#) objects to apply transformations to the mesh:

```
mesh = MeshVertexMerger()

# create your mesh
mesh.add_face(...)

# convert mesh to MeshTransformer object
return MeshTransformer.from_builder(mesh)
```

class ezdxf.render.**MeshVertexMerger** (*precision: int = 6*)

Subclass of [MeshBuilder](#)

Mesh with unique vertices and no doublets, but needs extra memory for bookkeeping.

[MeshVertexMerger](#) creates a key for every vertex by rounding its components by the Python `round()` function and a given *precision* value. Each vertex with the same key gets the same vertex index, which is the index of first vertex with this key, so all vertices with the same key will be located at the location of this first vertex. If you want an average location of and for all vertices with the same key look at the [MeshAverageVertexMerger](#) class.

Parameters **precision** – floating point precision for vertex rounding

6.10.10 MeshAverageVertexMerger

This is an extended version of [MeshVertexMerger](#). Location of merged vertices is the average location of all vertices with the same key, this needs extra memory and runtime in comparison to [MeshVertexMerger](#) and this

class also does not support transformations.

```
class ezdxf.render.MeshAverageVertexMerger (precision: int = 6)
    Subclass of MeshBuilder
```

Mesh with unique vertices and no doublets, but needs extra memory for bookkeeping and runtime for calculation of average vertex location.

[MeshAverageVertexMerger](#) creates a key for every vertex by rounding its components by the Python `round()` function and a given *precision* value. Each vertex with the same key gets the same vertex index, which is the index of first vertex with this key, the difference to the [MeshVertexMerger](#) class is the calculation of the average location for all vertices with the same key, this needs extra memory to keep track of the count of vertices for each key and extra runtime for updating the vertex location each time a vertex with an existing key is added.

Parameters `precision` – floating point precision for vertex rounding

6.10.11 Trace

This module provides tools to create banded lines like LWPOLYLINE with width information. Path rendering as quadrilaterals: [Trace](#), [Solid](#) or [Face3d](#).

```
class ezdxf.render.trace.TraceBuilder
```

Sequence of 2D banded lines like polylines with start- and end width or curves with start- and end width.

Note: Accepts 3D input, but z-axis is ignored. The [TraceBuilder](#) is a 2D only object and uses only the [OCS](#) coordinates!

abs_tol

Absolute tolerance for floating point comparisons

append(*trace: ezdxf.render.trace.AbstractTrace*) → None

Append a new trace.

close()

Close multi traces by merging first and last trace, if linear traces.

faces() → Iterable[Sequence[Vec2]]

Yields all faces as 4-tuples of [Vec2](#) objects in [OCS](#).

faces_wcs(*ocs: OCS, elevation: float*) → Iterable[Sequence[Vec3]]

Yields all faces as 4-tuples of [Vec3](#) objects in [WCS](#).

virtual_entities(*dxftype='TRACE', dxfattribs=None, doc: Drawing = None*) → Iterable[Union[Solid, Trace, Face3d]]

Yields faces as SOLID, TRACE or 3DFACE entities with DXF attributes given in *dxfattribs*.

If a document is given, the doc attribute of the new entities will be set and the new entities will be automatically added to the entity database of that document.

Note: The [TraceBuilder](#) is a 2D only object and uses only the [OCS](#) coordinates!

Parameters

- **dxftype** – DXF type as string, “SOLID”, “TRACE” or “3DFACE”
- **dxfattribs** – DXF attributes for SOLID, TRACE or 3DFACE entities

- **doc** – associated document

classmethod from polyline(polyline: DXFGraphic, segments: int = 64) → TraceBuilder
Create a complete trace from a LWPOLYLINE or a 2D POLYLINE entity, the trace consist of multiple sub-traces if *bulge* values are present. Uses only the *OCS* coordinates!

Parameters

- **polyline** – *LWPolyline* or 2D *Polyline*
- **segments** – count of segments for bulge approximation, given count is for a full circle, partial arcs have proportional less segments, but at least 3

__len__()

__getitem__(item)

class ezdxf.render.trace.LinearTrace

Linear 2D banded lines like polylines with start- and end width.

Accepts 3D input, but z-axis is ignored.

abs_tol

Absolute tolerance for floating point comparisons

is_started

True if at least one station exist.

add_station(point: Union[Sequence[float], ezdxf.math._vector.Vec2, ezdxf.math._vector.Vec3], start_width: float, end_width: float = None) → None

Add a trace station (like a vertex) at location *point*, *start_width* is the width of the next segment starting at this station, *end_width* is the end width of the next segment.

Adding the last location again, replaces the actual last location e.g. adding lines (a, b), (b, c), creates only 3 stations (a, b, c), this is very important to connect to/from splines.

Parameters

- **point** – 2D location (vertex), z-axis of 3D vertices is ignored.
- **start_width** – start width of next segment
- **end_width** – end width of next segment

faces() → Iterable[Tuple[Vec2, Vec2, Vec2, Vec2]]

Yields all faces as 4-tuples of *Vec2* objects.

First and last miter is 90 degrees if the path is not closed, otherwise the intersection of first and last segment is taken into account, a closed path has to have explicit the same last and first vertex.

virtual_entities(dxftype='TRACE', dxfattribs=None, doc: Drawing = None) → Iterable[Union[Solid, Trace, Face3d]]

Yields faces as SOLID, TRACE or 3DFACE entities with DXF attributes given in *dfattribs*.

If a document is given, the doc attribute of the new entities will be set and the new entities will be automatically added to the entity database of that document.

Parameters

- **dxftype** – DXF type as string, “SOLID”, “TRACE” or “3DFACE”
- **dfattribs** – DXF attributes for SOLID, TRACE or 3DFACE entities
- **doc** – associated document

class ezdxf.render.trace.CurvedTrace

2D banded curves like arcs or splines with start- and end width.

Represents always only one curved entity and all miter of curve segments are perpendicular to curve tangents.

Accepts 3D input, but z-axis is ignored.

faces () → Iterable[Tuple[Vec2, Vec2, Vec2, Vec2]]

Yields all faces as 4-tuples of `Vec2` objects.

virtual_entities (dxftype='TRACE', dxfattribs=None, doc: Drawing = None) → Iterable[Union[Solid, Trace, Face3d]]

Yields faces as SOLID, TRACE or 3DFACE entities with DXF attributes given in `dfattribs`.

If a document is given, the doc attribute of the new entities will be set and the new entities will be automatically added to the entity database of that document.

Parameters

- **dxftype** – DXF type as string, “SOLID”, “TRACE” or “3DFACE”
- **dfattribs** – DXF attributes for SOLID, TRACE or 3DFACE entities
- **doc** – associated document

classmethod from_arc (arc: ezdxf.math.arc.ConstructionArc, start_width: float, end_width: float, segments: int = 64) → ezdxf.render.trace.CurvedTrace

Create curved trace from an arc.

Parameters

- **arc** – `ConstructionArc` object
- **start_width** – start width
- **end_width** – end width
- **segments** – count of segments for full circle (360 degree) approximation, partial arcs have proportional less segments, but at least 3

Raises `ValueError` – if arc.radius <= 0

classmethod from_bspline (bspline: ezdxf.math.bspline.BSpline, start_width: float, end_width: float, segments: int) → ezdxf.render.trace.CurvedTrace

Create curved trace from a B-spline.

Parameters

- **bspline** – `BSpline` object
- **start_width** – start width
- **end_width** – end width
- **segments** – count of segments for approximation

6.10.12 Point Rendering

Helper function to render `Point` entities as DXF primitives.

ezdxf.render.point.virtual_entities (point: Point, pdszie: float = 1, pdmode: int = 0) → List[DXFGraphic]

Yields point graphic as DXF primitives LINE and CIRCLE entities. The dimensionless point is rendered as zero-length line!

Check for this condition:

```
e.dxftype() == 'LINE' and e.dxf.start.isclose(e.dxf.end)
```

if the rendering engine can't handle zero-length lines.

Parameters

- **point** – DXF POINT entity
- **pdszie** – point size in drawing units
- **pemode** – point styling mode, see *Point* class

New in version 0.15.

See also:

Go to `ezdxf.entities.Point` class documentation for more information about POINT styling modes.

6.10.13 MultiLeaderBuilder

New in version 0.18.

```
class ezdxf.render.MultiLeaderBuilder
```

Abstract base class to build *MultiLeader* entities.

```
class ezdxf.render.MultiLeaderMTextBuilder
```

Specialization of *MultiLeaderBuilder* to build *MultiLeader* with MTEXT content.

```
class ezdxf.render.MultiLeaderBlockBuilder
```

Specialization of *MultiLeaderBuilder* to build *MultiLeader* with BLOCK content.

6.10.14 Arrows

This module provides support for the AutoCAD standard arrow heads used in DIMENSION, LEADER and MULTI-LEADER entities.

TODO

6.11 Add-ons

6.11.1 Drawing / Export Addon

This add-on provides the functionality to render a DXF document to produce a rasterized or vector-graphic image which can be saved to a file or viewed interactively depending on the backend being used.

The module provides two example scripts in the folder `examples/addons/drawing` which can be run to save rendered images to files or view an interactive visualisation

Example for the usage of the `matplotlib` backend:

```
import sys
import matplotlib.pyplot as plt
from ezdxf import recover
from ezdxf.addons.drawing import RenderContext, Frontend
from ezdxf.addons.drawing.matplotlib import MatplotlibBackend
```

(continues on next page)

(continued from previous page)

```
# Safe loading procedure (requires ezdxf v0.14):
try:
    doc, auditor = recover.readfile('your.dxf')
except IOError:
    print(f'Not a DXF file or a generic I/O error.')
    sys.exit(1)
except ezdxf.DXFStructureError:
    print(f'Invalid or corrupted DXF file.')
    sys.exit(2)

# The auditor.errors attribute stores severe errors,
# which may raise exceptions when rendering.
if not auditor.has_errors:
    fig = plt.figure()
    ax = fig.add_axes([0, 0, 1, 1])
    ctx = RenderContext(doc)
    out = MatplotlibBackend(ax)
    Frontend(ctx, out).draw_layout(doc.modelspace(), finalize=True)
    fig.savefig('your.png', dpi=300)
```

Simplified render workflow but with less control:

```
from ezdxf import recover
from ezdxf.addons.drawing import matplotlib

# Exception handling left out for compactness:
doc, auditor = recover.readfile('your.dxf')
if not auditor.has_errors:
    matplotlib.qsave(doc.modelspace(), 'your.png')
```

MatplotlibBackend

```
class ezdxf.addons.drawing.matplotlib.MatplotlibBackend

    __init__(ax: plt.Axes, *, adjust_figure: bool = True, font: FontProperties, use_text_cache: bool =
        True)
```

PyQtBackend

```
class ezdxf.addons.drawing.pyqt.PyQtBackend

    __init__(scene: qw.QGraphicsScene = None, *, use_text_cache: bool = True, debug_draw_rect: bool =
        False)
```

Configuration

Additional options for the drawing add-on can be passed by the *config* argument of the `Frontend` constructor `__init__()`. Not every option will be supported by all backends.

```
class ezdxf.addons.drawing.config.Configuration
    Configuration options for the drawing add-on.
```

pdszie

the size to draw POINT entities (in drawing units) set to None to use the \$PDSIZE value from the dxf document header

0	5% of draw area height
<0	Specifies a percentage of the viewport size
>0	Specifies an absolute size
None	use the \$PDMODE value from the dxf document header

pdmode

point styling mode (see POINT documentation)

see [Point](#) class documentation

measurement

whether to use metric or imperial units as enum `ezdxf.enums.Measurement`

0	use imperial units (in, ft, yd, ...)
1	use metric units (ISO meters)
None	use the \$MEASUREMENT value from the dxf document header

show_defpoints

whether to show or filter out POINT entities on the defpoints layer

proxy_graphic_policy

the action to take when a proxy graphic is encountered

line_policy

the method to use when drawing styled lines (eg dashed, dotted etc)

hatch_policy

the method to use when drawing HATCH entities

infinite_line_length

the length to use when drawing infinite lines

lineweight_scaling

set to 0.0 for a constant minimal width the current result is correct, in SVG the line width is 0.7 points for 0.25mm as required, but it often looks too thick

min_lineweight

the minimum line width in 1/300 inch, set to None for let the the backend choose.

min_dash_length

the minimum length for a dash when drawing a styled line (default value is arbitrary)

max_flattening_distance

Max flattening distance in drawing units see Path.flattening documentation. The backend implementation should calculate an appropriate value, like 1 screen- or paper pixel on the output medium, but converted into drawing units. Sets Path() approximation accuracy

circle_approximation_count

Approximate a full circle by n segments, arcs have proportional less segments. Only used for approximation of arcs in banded polylines.

defaults()

Returns a frozen [Configuration](#) object with default values.

with_changes()

Returns a new frozen `Configuration` object with modified values.

Usage:

```
my_config = Configuration.defaults()
my_config = my_config.with_changes(lineweight_scaling=2)
```

LinePolicy

```
class ezdxf.addons.drawing.config.LinePolicy
```

SOLID

draw all lines as solid regardless of the linetype style

APPROXIMATE

use the closest approximation available to the backend for rendering styled lines

ACCURATE

analyse and render styled lines as accurately as possible. This approach is slower and is not well suited to interactive applications.

HatchPolicy

```
class ezdxf.addons.drawing.config.HatchPolicy
```

The action to take when a HATCH entity is encountered

IGNORE

do not show HATCH entities at all

SHOW_OUTLINE

show only the outline of HATCH entities

SHOW_SOLID

show HATCH entities but draw with solid fill regardless of the pattern

SHOW_APPROXIMATE_PATTERN

show HATCH entities using the closest approximation available to the current backend

ProxyGraphicPolicy

```
class ezdxf.addons.drawing.config.ProxyGraphicPolicy
```

The action to take when an entity with a proxy graphic is encountered

Note: To get proxy graphics support proxy graphics have to be loaded: Set the global option `ezdxf.options.load_proxy_graphics` to True, which is the default value.

This can not prevent drawing proxy graphic inside of blocks, because this is outside of the domain of the drawing add-on!

IGNORE

do not display proxy graphics (skip_entity will be called instead)

SHOW

if the entity cannot be rendered directly (eg if not implemented) but a proxy is present: display the proxy

PREFER

display proxy graphics even for entities where direct rendering is available

Properties

```
class ezdxf.addons.drawing.properties.Properties
```

LayerProperties

```
class ezdxf.addons.drawing.properties.LayerProperties
```

RenderContext

```
class ezdxf.addons.drawing.properties.RenderContext
```

Frontend

```
class ezdxf.addons.drawing.frontend.Frontend
```

Backend

```
class ezdxf.addons.drawing.backend.Backend
```

Details

The rendering is performed in two stages. The front-end traverses the DXF document structure, converting each encountered entity into primitive drawing commands. These commands are fed to a back-end which implements the interface: *Backend*.

Currently a `PyQtBackend` (`QGraphicsScene` based) and a `MatplotlibBackend` are implemented.

Although the resulting images will not be pixel-perfect with AutoCAD (which was taken as the ground truth when developing this add-on) great care has been taken to achieve similar behavior in some areas:

- The algorithm for determining color should match AutoCAD. However, the color palette is not stored in the dxf file, so the chosen colors may be different to what is expected. The `RenderContext` class supports passing a plot style table (`CTB`-file) as custom color palette but uses the same palette as AutoCAD by default.
- Text rendering is quite accurate, text positioning, alignment and word wrapping are very faithful. Differences may occur if a different font from what was used by the CAD application but even in that case, for supported backends, measurements are taken of the font being used to match text as closely as possible.
- Visibility determination (based on which layers are visible) should match AutoCAD

See `examples/addons/drawing/cad_viewer.py` for an advanced use of the module.

See `examples/addons/drawing/draw_cad.py` for a simple use of the module.

See `drawing.md` in the `ezdxf` repository for additional behaviours documented during the development of this add-on.

Limitations

- Rich text formatting is ignored (drawn as plain text)
- If the backend does not match the font then the exact text placement and wrapping may appear slightly different
- MULTILEADER renders only proxy graphic if available
- relative size of POINT entities cannot be replicated exactly
- only basic support for:
 - infinite lines (rendered as lines with a finite length)
 - VIEWPORT and OLE2FRAME entities (rendered as rectangles)
 - 3D entities are projected into the xy-plane and 3D text is not supported
 - vertical text (will render as horizontal text)
 - multiple columns of text (placement of additional columns may be incorrect)

6.11.2 Geo Interface

Intended Usage

The intended usage of the `ezdxf.addons.geo` module is as tool to work with geospatial data in conjunction with dedicated geospatial applications and libraries and the module can not and should not replicate their functionality.

The only reimplemented feature is the most common WSG84 EPSG:3395 World Mercator projection, for everything else use the dedicated packages like:

- [pyproj](#) - Cartographic projections and coordinate transformations library.
- [Shapely](#) - Manipulation and analysis of geometric objects in the Cartesian plane.
- [PyShp](#) - The Python Shapefile Library (PyShp) reads and writes ESRI Shapefiles in pure Python.
- [GeoJSON](#) - GeoJSON interface for Python.
- [GDAL](#) - Tools for programming and manipulating the GDAL Geospatial Data Abstraction Library.
- [Fiona](#) - Fiona is GDAL's neat and nimble vector API for Python programmers.
- [QGIS](#) - A free and open source geographic information system.
- and many more ...

This module provides support for the `__geo_interface__`: <https://gist.github.com/sgillies/2217756>

Which is also supported by [Shapely](#), for supported types see the [GeoJSON](#) Standard and examples in [Appendix-A](#).

See also:

[Tutorial for the Geo Add-on](#) for loading GPX data into DXF files with an existing geo location reference and exporting DXF entities as GeoJSON data.

Proxy From Mapping

The `GeoProxy` represents a `__geo_interface__` mapping, create a new proxy by `GeoProxy.parse()` from an external `__geo_interface__` mapping. `GeoProxy.to_dxf_entities()` returns new DXF entities from this mapping. Returns “Point” as `Point` entity, “LineString” as `LWPolyline` entity and “Polygon” as `Hatch` entity

or as separated `LWPolyline` entities (or both) and new in v0.16.6 as `MPolygon`. Supports “MultiPoint”, “MultiLineString”, “MultiPolygon”, “GeometryCollection”, “Feature” and “FeatureCollection”. Add new DXF entities to a layout by the `Layout.add_entity()` method.

Proxy From DXF Entity

The `proxy()` function or the constructor `GeoProxy.from_dxf_entities()` creates a new `GeoProxy` object from a single DXF entity or from an iterable of DXF entities, entities without a corresponding representation will be approximated.

Supported DXF entities are:

- POINT as “Point”
- LINE as “LineString”
- LWPOLYLINE as “LineString” if open and “Polygon” if closed
- POLYLINE as “LineString” if open and “Polygon” if closed, supports only 2D and 3D polylines, POLYMESH and POLYFACE are not supported
- SOLID, TRACE, 3DFACE as “Polygon”
- CIRCLE, ARC, ELLIPSE and SPLINE by approximation as “LineString” if open and “Polygon” if closed
- HATCH and MPOLYGON as “Polygon”, holes are supported

New in version 0.16.6: MPOLYGON support

Warning: This module does no extensive validity checks for “Polygon” objects and because DXF has different requirements for HATCH boundary paths than the `GeoJSON` Standard, it is possible to create invalid “Polygon” objects. It is recommended to check critical objects by a sophisticated geometry library like `Shapely`.

Module Functions

```
ezdxf.addons.geo.proxy(entity: Union[DXFGraphic, Iterable[DXFGraphic]], distance=0.1,  
force_line_string=False) → GeoProxy
```

Returns a `GeoProxy` object.

Parameters

- **entity** – a single DXF entity or iterable of DXF entities
- **distance** – maximum flattening distance for curve approximations
- **force_line_string** – by default this function returns Polygon objects for closed geometries like CIRCLE, SOLID, closed POLYLINE and so on, by setting argument `force_line_string` to True, this entities will be returned as LineString objects.

```
ezdxf.addons.geo.dxf_entities(geo_mapping, polygon=1, dxfattribs: Dict = None) → Iterable[DXFGraphic]
```

Returns __geo_interface__ mappings as DXF entities.

The `polygon` argument determines the method to convert polygons, use 1 for `Hatch` entity, 2 for `LWPolyline` or 3 for both. Option 2 returns for the exterior path and each hole a separated `LWPolyline` entity. The `Hatch` entity supports holes, but has no explicit border line.

Yields `Hatch` always before `LWPolyline` entities.

`MPolygon` support was added in v0.16.6, which is like a `Hatch` entity with additional border lines, but the MPOLYGON entity is not a core DXF entity and DXF viewers, applications and libraries may not support this entity. The DXF attribute `color` defines the border line color and `fill_color` the color of the solid filling.

The returned DXF entities can be added to a layout by the `Layout.add_entity()` method.

Parameters

- `geo_mapping` – `__geo_interface__` mapping as dict or a Python object with a `__geo_interface__` property
- `polygon` – method to convert polygons (1-2-3-4)
- `dxfattribs` – dict with additional DXF attributes

`ezdxf.addons.geo.gfilter(entities: Iterable[DXFGraphic]) → Iterable[DXFGraphic]`

Filter DXF entities from iterable `entities`, which are incompatible to the `__geo_reference__` interface.

GeoProxy Class

`class ezdxf.addons.geo.GeoProxy(geo_mapping: Dict[KT, VT], places: int = 6)`

Stores the `__geo_interface__` mapping in a parsed and compiled form.

Stores coordinates as `Vec3` objects and represents “Polygon” always as tuple (exterior, holes) even without holes.

The GeoJSON specification recommends 6 decimal places for latitude and longitude which equates to roughly 10cm of precision. You may need slightly more for certain applications, 9 decimal places would be sufficient for professional survey-grade GPS coordinates.

Parameters

- `geo_mapping` – parsed and compiled `__geo_interface__` mapping
- `places` – decimal places to round for `__geo_interface__` export

`__geo_interface__`

Returns the `__geo_interface__` compatible mapping as dict.

`geotype`

Property returns the top level entity type or None.

`classmethod parse(geo_mapping: Dict) → GeoProxy`

Parse and compile a `__geo_interface__` mapping as dict or a Python object with a `__geo_interface__` property, does some basic syntax checks, converts all coordinates into `Vec3` objects, represents “Polygon” always as tuple (exterior, holes) even without holes.

`classmethod from_dxf_entities(entity: Union[DXFGraphic, Iterable[DXFGraphic]], distance=0.1, force_line_string=False) → GeoProxy`

Constructor from a single DXF entity or an iterable of DXF entities.

Parameters

- `entity` – DXF entity or entities
- `distance` – maximum flattening distance for curve approximations
- `force_line_string` – by default this function returns Polygon objects for closed geometries like CIRCLE, SOLID, closed POLYLINE and so on, by setting argument `force_line_string` to True, this entities will be returned as LineString objects.

to_dxf_entities (*polygon*=1, *dxfattribs*: *Dict* = *None*) → Iterable[DXFGraphic]

Returns stored `__geo_interface__` mappings as DXF entities.

The *polygon* argument determines the method to convert polygons, use 1 for `Hatch` entity, 2 for `LWPolyline` or 3 for both. Option 2 returns for the exterior path and each hole a separated LWPolyline entity. The Hatch entity supports holes, but has no explicit border line.

Yields Hatch always before LWPolyline entities.

`MPolygon` support was added in v0.16.6, which is like a `Hatch` entity with additional border lines, but the MPOLYGON entity is not a core DXF entity and DXF viewers, applications and libraries may not support this entity. The DXF attribute *color* defines the border line color and *fill_color* the color of the solid filling.

The returned DXF entities can be added to a layout by the `Layout.add_entity()` method.

Parameters

- **polygon** – method to convert polygons (1-2-3-4)
- **dxfattribs** – dict with additional DXF attributes

New in version 0.16.6: MPOLYGON support

copy() → GeoProxy

Returns a deep copy.

`__iter__()` → Iterable[Dict[KT, VT]]

Iterate over all geo content objects.

Yields only “Point”, “LineString”, “Polygon”, “MultiPoint”, “MultiLineString” and “MultiPolygon” objects, returns the content of “GeometryCollection”, “FeatureCollection” and “Feature” as geometry objects (“Point”, ...).

wcs_to_crs (*crs*: *ezdxf.math._matrix44.Matrix44*) → None

Transform all coordinates recursive from `WCS` coordinates into Coordinate Reference System (CRS) by transformation matrix *crs* inplace.

The CRS is defined by the `GeoData` entity, get the `GeoData` entity from the modelspace by method `get_geodata()`. The CRS transformation matrix can be acquired from the `GeoData` object by `get_crs_transformation()` method:

```
doc = eздxf.readfile('file.dxf')
msp = doc.modelspace()
geodata = msp.get_geodata()
if geodata:
    matrix, axis_ordering = geodata.get_crs_transformation()
```

If *axis_ordering* is `False` the CRS is not compatible with the `__geo_interface__` or GeoJSON (see chapter 3.1.1).

Parameters **crs** – transformation matrix of type `Matrix44`

crs_to_wcs (*crs*: *ezdxf.math._matrix44.Matrix44*) → None

Transform all coordinates recursive from CRS into `WCS` coordinates by transformation matrix *crs* inplace, see also `GeoProxy.wcs_to_crs()`.

Parameters **crs** – transformation matrix of type `Matrix44`

globe_to_map (*func*: *Callable*[*Vec3*, *Vec3*] = *None*) → None

Transform all coordinates recursive from globe representation in longitude and latitude in decimal degrees into 2D map representation in meters.

Default is WGS84 EPSG:4326 (GPS) to WGS84 EPSG:3395 World Mercator function `wgs84_4326_to_3395()`.

Use the `pyproj` package to write a custom projection function as needed.

Parameters `func` – custom transformation function, which takes one `Vec3` object as argument and returns the result as a `Vec3` object.

map_to_globe (`func: Callable[[Vec3], Vec3] = None`) → None

Transform all coordinates recursive from 2D map representation in meters into globe representation as longitude and latitude in decimal degrees.

Default is WGS84 EPSG:3395 World Mercator to WGS84 EPSG:4326 GPS function `wgs84_3395_to_4326()`.

Use the `pyproj` package to write a custom projection function as needed.

Parameters `func` – custom transformation function, which takes one `Vec3` object as argument and returns the result as a `Vec3` object.

apply (`func: Callable[[Vec3], Vec3]`) → None

Apply the transformation function `func` recursive to all coordinates.

Parameters `func` – transformation function as `Callable[[Vec3], Vec3]`

filter (`func: Callable[[GeoProxy], bool]`) → None

Removes all mappings for which `func()` returns `False`. The function only has to handle Point, LineString and Polygon entities, other entities like MultiPolygon are divided into separate entities also any collection.

Helper Functions

`ezdxf.addons.geo.wgs84_4326_to_3395(location: Vec3)` → `Vec3`

Transform WGS84 EPSG:4326 location given as latitude and longitude in decimal degrees as used by GPS into World Mercator cartesian 2D coordinates in meters EPSG:3395.

Parameters `location` – `Vec3` object, x-attribute represents the longitude value (East-West) in decimal degrees and the y-attribute represents the latitude value (North-South) in decimal degrees.

`ezdxf.addons.geo.wgs84_3395_to_4326(location: Vec3, tol=1e-6)` → `Vec3`

Transform WGS84 World Mercator EPSG:3395 location given as cartesian 2D coordinates x, y in meters into WGS84 decimal degrees as longitude and latitude EPSG:4326 as used by GPS.

Parameters

- `location` – `Vec3` object, z-axis is ignored
- `tol` – accuracy for latitude calculation

`ezdxf.addons.geo.dms2dd(d: float, m: float = 0, s: float = 0)` → `float`

Convert degree, minutes, seconds into decimal degrees.

`ezdxf.addons.geo.dd2dms(dd: float)` → `Tuple[float, float, float]`

Convert decimal degrees into degree, minutes, seconds.

6.11.3 Importer

This add-on is meant to import graphical entities from another DXF drawing and their required table entries like LAYER, LTYPE or STYLE.

Because of complex extensibility of the DXF format and the lack of sufficient documentation, I decided to remove most of the possible source drawing dependencies from imported entities, therefore imported entities may not look the same as the original entities in the source drawing, but at least the geometry should be the same and the DXF file does not break.

Removed data which could contain source drawing dependencies: Extension Dictionaries, AppData and XDATA.

Warning: DON'T EXPECT PERFECT RESULTS!

The *Importer* supports following data import:

- entities which are really safe to import: LINE, POINT, CIRCLE, ARC, TEXT, SOLID, TRACE, 3DFACE, SHAPE, POLYLINE, ATTRIB, ATTDEF, INSERT, ELLIPSE, MTEXT, LWPOLYLINE, SPLINE, HATCH, MESH, XLINE, RAY, DIMENSION, LEADER, VIEWPORT
- table and table entry import is restricted to LAYER, LTYPE, STYLE, DIMSTYLE
- import of BLOCK definitions is supported
- import of paper space layouts is supported

Import of DXF objects from the OBJECTS section is not supported.

DIMSTYLE override for entities DIMENSION and LEADER is not supported.

Example:

```
import ezdxf
from ezdxf.addons import Importer

sdoc = ezdxf.readfile('original.dxf')
tdoc = ezdxf.new()

importer = Importer(sdoc, tdoc)

# import all entities from source modelspace into modelspace of the target drawing
importer.import_modelsheets()

# import all paperspace layouts from source drawing
importer.import_paperspace_layouts()

# import all CIRCLE and LINE entities from source modelspace into an arbitrary target_
# layout.
# create target layout
tblock = tdoc.blocks.new('SOURCE_ENTS')
# query source entities
ents = sdoc.modelsheet().query('CIRCLE LINE')
# import source entities into target block
importer.import_entities(ents, tblock)

# This is ALWAYS the last & required step, without finalizing the target drawing is_
# maybe invalid!
# This step imports all additional required table entries and block definitions.
importer.finalize()

tdoc.saveas('imported.dxf')
```

class ezdxf.addons.importer.**Importer**(source: Drawing, target: Drawing)

The *Importer* class is central element for importing data from other DXF documents.

Parameters

- **source** – source Drawing
- **target** – target Drawing

source

source DXF document

target

target DXF document

used_layers

Set of used layer names as string, AutoCAD accepts layer names without a LAYER table entry.

used_linetypes

Set of used linetype names as string, these linetypes require a TABLE entry or AutoCAD will crash.

used_styles

Set of used text style names, these text styles require a TABLE entry or AutoCAD will crash.

used_dimstyles

Set of used dimension style names, these dimension styles require a TABLE entry or AutoCAD will crash.

finalize() → None

Finalize the import by importing required table entries and BLOCK definitions, without finalization the target document is maybe invalid for AutoCAD. Call the `finalize()` method as last step of the import process.

import_block(block_name: str, rename=True) → str

Import one BLOCK definition from source document.

If the BLOCK already exist the BLOCK will be renamed if argument `rename` is True, otherwise the existing BLOCK in the target document will be used instead of the BLOCK in the source document. Required name resolving for imported block references (INSERT), will be done in the `Importer.finalize()` method.

To replace an existing BLOCK in the target document, just delete it before importing data: `target.blocks.delete_block(block_name, safe=False)`

Parameters

- **block_name** – name of BLOCK to import
- **rename** – rename BLOCK if a BLOCK with the same name already exist in target document

Returns: (renamed) BLOCK name

Raises `ValueError` – BLOCK in source document not found (defined)

import_blocks(block_names: Iterable[str], rename=False) → None

Import all BLOCK definitions from source document.

If a BLOCK already exist the BLOCK will be renamed if argument `rename` is True, otherwise the existing BLOCK in the target document will be used instead of the BLOCK from the source document. Required name resolving for imported BLOCK references (INSERT), will be done in the `Importer.finalize()` method.

Parameters

- **block_names** – names of BLOCK definitions to import
- **rename** – rename BLOCK if a BLOCK with the same name already exist in target document

Raises `ValueError` – BLOCK in source document not found (defined)

import_entities (`entities: Iterable[DXFEntity]`, `target_layout: BaseLayout = None`) → `None`

Import all `entities` into `target_layout` or the modelspace of the target document, if `target_layout` is `None`.

Parameters

- `entities` – Iterable of DXF entities
- `target_layout` – any layout (modelspace, paperspace or block) from the target document

Raises `DXFStructureError` – `target_layout` is not a layout of target document

import_entity (`entity: DXFEntity`, `target_layout: BaseLayout = None`) → `None`

Imports a single DXF `entity` into `target_layout` or the modelspace of the target document, if `target_layout` is `None`.

Parameters

- `entity` – DXF entity to import
- `target_layout` – any layout (modelspace, paperspace or block) from the target document

Raises `DXFStructureError` – `target_layout` is not a layout of target document

import_models (`target_layout: BaseLayout = None`) → `None`

Import all entities from source modelspace into `target_layout` or the modelspace of the target document, if `target_layout` is `None`.

Parameters `target_layout` – any layout (modelspace, paperspace or block) from the target document

Raises `DXFStructureError` – `target_layout` is not a layout of target document

import_paperspace_layout (`name: str`) → `Layout`

Import paperspace layout `name` into the target document.

Recreates the source paperspace layout in the target document, renames the target paperspace if a paperspace with same `name` already exist and imports all entities from the source paperspace into the target paperspace.

Parameters `name` – source paper space name as string

Returns: new created target paperspace Layout

Raises

- `KeyError` – source paperspace does not exist
- `DXFTypError` – invalid modelspace import

import_paperspace_layouts () → `None`

Import all paperspace layouts and their content into the target document. Target layouts will be renamed if a layout with the same name already exist. Layouts will be imported in original tab order.

import_shape_files (`fonts: Set[str]`) → `None`

Import shape file table entries from the source document into the target document. Shape file entries are stored in the styles table but without a name.

import_table (`name: str, entries: Union[str, Iterable[str]] = '*', replace=False`) → `None`

Import specific table entries from the source document into the target document.

Parameters

- **name** – valid table names are “layers”, “linetypes” and “styles”
- **entries** – Iterable of table names as strings, or a single table name or “*” for all table entries
- **replace** – True to replace the already existing table entry else ignore existing entries

Raises `TypeError` – unsupported table type

```
import_tables(table_names: Union[str, Iterable[str]] = '*', replace=False) → None
Import DXF tables from the source document into the target docuemnt.
```

Parameters

- **table_names** – iterable of tables names as strings, or a single table name as string or “*” for all supported tables
- **replace** – True to replace already existing table entries else ignore existing entries

Raises `TypeError` – unsupported table type

```
recreate_source_layout(name: str) → Layout
```

Recreate source paperspace layout *name* in the target document. The layout will be renamed if *name* already exist in the target document. Returns target modelspace for layout name “Model”.

Parameters **name** – layout name as string

Raises `KeyError` – if source layout *name* not exist

6.11.4 dxf2code

Translate DXF entities and structures into Python source code.

Short example:

```
import ezdxf
from ezdxf.addons.dxf2code import entities_to_code, block_to_code

doc = ezdxf.readfile('original.dxf')
msp = doc.modelspace()
source = entities_to_code(msp)

# create source code for a block definition
block_source = block_to_code(doc.blocks['MyBlock'])

# merge source code objects
source.merge(block_source)

with open('source.py', mode='wt') as f:
    f.write(source.import_str())
    f.write('\n\n')
    f.write(source.code_str())
    f.write('\n')
```

`ezdxf.addons.dxf2code.entities_to_code(entities: Iterable[DXFEntity], layout: str = 'layout', ignore: Iterable[str] = None) → Code`

Translates DXF entities into Python source code to recreate this entities by ezdxf.

Parameters

- **entities** – iterable of DXFEntity

- **layout** – variable name of the layout (model space or block) as string
- **ignore** – iterable of entities types to ignore as strings like ['IMAGE', 'DIMENSION']

Returns `Code`

```
ezdxf.addons.dxf2code.block_to_code(block: BlockLayout, drawing: str = 'doc', ignore: Iterable[str] = None) → Code
```

Translates a BLOCK into Python source code to recreate the BLOCK by eздxf.

Parameters

- **block** – block definition layout
- **drawing** – variable name of the drawing as string
- **ignore** – iterable of entities types to ignore as strings like ['IMAGE', 'DIMENSION']

Returns `Code`

```
ezdxf.addons.dxf2code.table_entries_to_code(entities: Iterable[DXFEntity], drawing='doc') → Code
```

```
ezdxf.addons.dxf2code.black(code: str, line_length=88, fast: bool = True) → str
```

Returns the source `code` as a single string formatted by `Black`

Requires the installed `Black` formatter:

```
pip3 install black
```

Parameters

- **code** – source code
- **line_length** – max. source code line length
- **fast** – True for fast mode, False to check that the reformatted code is valid

Raises `ImportError` – `Black` is not available

```
class eздxf.addons.dxf2code.Code
```

Source code container.

code

Source code line storage, store lines without line ending \\n

imports

source code line storage for global imports, store lines without line ending \\n

layers

Layers used by the generated source code, AutoCAD accepts layer names without a LAYER table entry.

linetypes

Linetypes used by the generated source code, these linetypes require a TABLE entry or AutoCAD will crash.

styles

Text styles used by the generated source code, these text styles require a TABLE entry or AutoCAD will crash.

dimstyles

Dimension styles used by the generated source code, these dimension styles require a TABLE entry or AutoCAD will crash.

blocks

Blocks used by the generated source code, these blocks require a BLOCK definition in the BLOCKS section or AutoCAD will crash.

code_str (indent: int = 0) → str

Returns the source code as a single string.

Parameters `indent` – source code indentation count by spaces

block_code_str (line_length=88) → str

Returns the source code as a single string formatted by Black

Parameters `line_length` – max. source code line length

Raises `ImportError` – Black is not available

import_str (indent: int = 0) → str

Returns required imports as a single string.

Parameters `indent` – source code indentation count by spaces

merge (code: ezdxf.addons.dxf2code.Code, indent: int = 0) → None

Add another `Code` object.

add_import (statement: str) → None

Add import statement, identical import statements are merged together.

add_line (code: str, indent: int = 0) → None

Add a single source code line without line ending \n.

add_lines (code: Iterable[str], indent: int = 0) → None

Add multiple source code lines without line ending \n.

6.11.5 iterdxf

This add-on allows iterating over entities of the modelspace of really big (> 5GB) DXF files which do not fit into memory by only loading one entity at the time. Only ASCII DXF files are supported.

The entities are regular `DXFGraphic` objects with access to all supported DXF attributes, this entities can be written to new DXF files created by the `IterDXF.export()` method. The new `add_foreign_entity()` method allows also to add this entities to new regular `ezdxf` drawings (except for the INSERT entity), but resources like linetype and style are removed, only layer will be preserved but only with default attributes like color 7 and linetype CONTINUOUS.

The following example shows how to split a big DXF files into several separated DXF files which contains only LINE, TEXT or POLYLINE entities.

```
from ezdxf.addons import iterdxf

doc = iterdxf.opendxf('big.dxf')
line_exporter = doc.export('line.dxf')
text_exporter = doc.export('text.dxf')
polyline_exporter = doc.export('polyline.dxf')
try:
    for entity in doc.modelspace():
        if entity.dxftype() == 'LINE':
            line_exporter.write(entity)
        elif entity.dxftype() == 'TEXT':
            text_exporter.write(entity)
        elif entity.dxftype() == 'POLYLINE':
```

(continues on next page)

(continued from previous page)

```
    polyline_exporter.write(entity)
finally:
    line_exporter.close()
    text_exporter.close()
    polyline_exporter.close()
    doc.close()
```

Supported DXF types:

3DFACE, ARC, ATTDEF, ATTRIB, CIRCLE, DIMENSION, ELLIPSE, HATCH, HELIX, IMAGE, INSERT, LEADER, LINE, LWPOLYLINE, MESH, MLEADER, MLINE, MTEXT, POINT, POLYLINE, RAY, SHAPE, SOLID, SPLINE, TEXT, TRACE, VERTEX, WIPEOUT, XLINE

Transfer simple entities to another DXF document, this works for some supported entities, except for entities with strong dependencies to the original document like INSERT look at `add_foreign_entity()` for all supported types:

```
newdoc = ezdxf.new()
msp = newdoc.modelspace()
# line is an entity from a big source file
msp.add_foreign_entity(line)
# and so on ...
msp.add_foreign_entity(lwpolyline)
msp.add_foreign_entity(mesh)
msp.add_foreign_entity(polyface)
```

Transfer MESH and POLYFACE (dxftype for POLYFACE and POLYMESH is POLYLINE!) entities into a new DXF document by the MeshTransformer class:

```
from ezdxf.render import MeshTransformer

# mesh is MESH from a big source file
t = MeshTransformer.from_mesh(mesh)
# create a new MESH entity from MeshTransformer
t.render(msp)

# polyface is POLYFACE from a big source file
t = MeshTransformer.from_polyface(polyface)
# create a new POLYMESH entity from MeshTransformer
t.render_polyface(msp)
```

Another way to import entities from a big source file into new DXF documents is to split the big file into smaller parts and use the `Importer` add-on for a more safe entity import.

`ezdxf.addons.iterdxf.opendxf(filename: str, errors: str='surrogateescape')` → IterDXF

Open DXF file for iterating, be sure to open valid DXF files, no DXF structure checks will be applied.

Use this function to split up big DXF files as shown in the example above.

Parameters

- **filename** – DXF filename of a seekable DXF file.
- **errors** – specify decoding error handler
 - “surrogateescape” to preserve possible binary data (default)
 - “ignore” to use the replacement char U+FFFD “” for invalid data
 - “strict” to raise an `UnicodeDecodeError` exception for invalid data

Raises

- `DXFStructureError` – invalid or incomplete DXF file
- `UnicodeDecodeError` – if `errors` is “strict” and a decoding error occurs

```
ezdxf.addons.iterdxf.modelspace(filename: str, types: Iterable[str]=None, errors: str='surrogateescape') → Iterable[DXFGraphic]
```

Iterate over all modelspace entities as `DXFGraphic` objects of a seekable file.

Use this function to iterate “quick” over modelspace entities of a DXF file, filtering DXF types may speed up things if many entity types will be skipped.

Parameters

- `filename` – filename of a seekable DXF file
- `types` – DXF types like `['LINE', '3DFACE']` which should be returned, `None` returns all supported types.
- `errors` – specify decoding error handler
 - “surrogateescape” to preserve possible binary data (default)
 - “ignore” to use the replacement char U+FFFD “” for invalid data
 - “strict” to raise an `UnicodeDecodeError` exception for invalid data

Raises

- `DXFStructureError` – invalid or incomplete DXF file
- `UnicodeDecodeError` – if `errors` is “strict” and a decoding error occurs

```
ezdxf.addons.iterdxf.single_pass_modelsapce(stream: BinaryIO, types: Iterable[str]=None, errors: str='surrogateescape') → Iterable[DXFGraphic]
```

Iterate over all modelspace entities as `DXFGraphic` objects in one single pass.

Use this function to ‘quick’ iterate over modelspace entities of a **not** seekable binary DXF stream, filtering DXF types may speed up things if many entity types will be skipped.

Parameters

- `stream` – (not seekable) binary DXF stream
- `types` – DXF types like `['LINE', '3DFACE']` which should be returned, `None` returns all supported types.
- `errors` – specify decoding error handler
 - “surrogateescape” to preserve possible binary data (default)
 - “ignore” to use the replacement char U+FFFD “” for invalid data
 - “strict” to raise an `UnicodeDecodeError` exception for invalid data

Raises

- `DXFStructureError` – Invalid or incomplete DXF file
- `UnicodeDecodeError` – if `errors` is “strict” and a decoding error occurs

```
class ezdxf.addons.ITERDXF
```

export (*name*: str) → IterDXFWriter

Returns a companion object to export parts from the source DXF file into another DXF file, the new file will have the same HEADER, CLASSES, TABLES, BLOCKS and OBJECTS sections, which guarantees all necessary dependencies are present in the new file.

Parameters **name** – filename, no special requirements

modelspace (*types*: Iterable[str] = *None*) → Iterable[DXFGraphic]

Returns an iterator for all supported DXF entities in the modelspace. These entities are regular *DXFGraphic* objects but without a valid document assigned. It is **not** possible to add these entities to other *ezdxf* documents.

It is only possible to recreate the objects by factory functions base on attributes of the source entity. For MESH, POLYMESH and POLYFACE it is possible to use the *MeshTransformer* class to render (recreate) this objects as new entities in another document.

Parameters **types** – DXF types like ['LINE', '3DFACE'] which should be returned,
None returns all supported types.

close()

Safe closing source DXF file.

class eздxf.addons.итердхф.ИтерДХФWriter**write** (*entity*: DXFGraphic)

Write a DXF entity from the source DXF file to the export file.

Don't write entities from different documents than the source DXF file, dependencies and resources will not match, maybe it will work once, but not in a reliable way for different DXF documents.

close()

Safe closing of exported DXF file. Copying of OBJECTS section happens only at closing the file, without closing the new DXF file is invalid.

6.11.6 r12writer

The fast file/stream writer creates simple DXF R12 drawings with just an ENTITIES section. The HEADER, TABLES and BLOCKS sections are not present except FIXED-TABLES are written. Only LINE, CIRCLE, ARC, TEXT, POINT, SOLID, 3DFACE and POLYLINE entities are supported. FIXED-TABLES is a predefined TABLES section, which will be written, if the init argument *fixed_tables* of *R12FastStreamWriter* is True.

The *R12FastStreamWriter* writes the DXF entities as strings direct to the stream without creating an in-memory drawing and therefore the processing is very fast.

Because of the lack of a BLOCKS section, BLOCK/INSERT can not be used. Layers can be used, but this layers have a default setting color = 7 (black/white) and linetype = 'Continuous'. If writing the FIXED-TABLES, some predefined text styles and line types are available, else text style is always 'STANDARD' and line type is always 'ByLayer'.

If using FIXED-TABLES, following predefined line types are available:

- CONTINUOUS
- CENTER _____
- CENTERX2 _____
- CENTER2 _____
- DASHED _____

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

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

- OpenSans
- OpenSansCondensed-Light

Tutorial

A simple example with different DXF entities:

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

with r12writer("quick_and_dirty_dxf_r12.dxf") as dxf:
    dxf.add_line((0, 0), (17, 23))
    dxf.add_circle((0, 0), radius=2)
    dxf.add_arc((0, 0), radius=3, start=0, end=175)
    dxf.add_solid([(0, 0), (1, 0), (0, 1), (1, 1)])
    dxf.add_point((1.5, 1.5))

    # 2d polyline, new in v0.12
    dxf.add_polyline_2d([(5, 5), (7, 3), (7, 6)])

    # 2d polyline with bulge value, new in v0.12
    dxf.add_polyline_2d([(5, 5), (7, 3, 0.5), (7, 6)], format='xyb')

    # 3d polyline only, changed in v0.12
    dxf.add_polyline([(4, 3, 2), (8, 5, 0), (2, 4, 9)])

    dxf.add_text("test the text entity", align="MIDDLE_CENTER")
```

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

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

MAX_X_COORD = 1000.0
MAX_Y_COORD = 1000.0
CIRCLE_COUNT = 1000000

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

Show all available line types:

```
import ezdxf

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

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

Reference

`ezdxf.addons.r12writer.r12writer(stream: Union[TextIO, BinaryIO, str], fixed_tables = False, fmt = 'asc')` → R12FastStreamWriter

Context manager for writing DXF entities to a stream/file. `stream` can be any file like object with a `write()` method or just a string for writing DXF entities to the file system. If `fixed_tables` is True, a standard TABLES section is written in front of the ENTITIES section and some predefined text styles and line types can be used.

Set argument `fmt` to “asc” to write ASCII DXF file (default) or “bin” to write Binary DXF files. ASCII DXF require a TextIO stream and Binary DXF require a BinaryIO stream.

`class ezdxf.addons.r12writer.R12FastStreamWriter(stream: TextIO, fixed_tables=False)`
Fast stream writer to create simple DXF R12 drawings.

Parameters

- `stream` – a file like object with a `write()` method.
- `fixed_tables` – if `fixed_tables` is True, a standard TABLES section is written in front of the ENTITIES section and some predefined text styles and line types can be used.

`close()` → None

Writes the DXF tail. Call is not necessary when using the context manager `r12writer()`.

`add_line(start: Sequence[float], end: Sequence[float], layer: str = '0', color: int = None, linetype: str = None)` → None
Add a LINE entity from `start` to `end`.

Parameters

- `start` – start vertex as `(x, y[, z])` tuple
- `end` – end vertex as as `(x, y[, z])` tuple

- **layer** – layer name as string, without a layer definition the assigned color = 7 (black/white) and line type is 'Continuous'.
- **color** – color as *AutoCAD Color Index (ACI)* in the range from 0 to 256, 0 is *ByBlock* and 256 is *ByLayer*, default is *ByLayer* which is always color = 7 (black/white) without a layer definition.
- **linetype** – line type as string, if FIXED-TABLES are written some predefined line types are available, else line type is always *ByLayer*, which is always 'Continuous' without a LAYERS table.

add_circle (*center*: Sequence[float], *radius*: float, *layer*: str = '0', *color*: int = None, *linetype*: str = None) → None
Add a CIRCLE entity.

Parameters

- **center** – circle center point as (x, y) tuple
- **radius** – circle radius as float
- **layer** – layer name as string see [add_line\(\)](#)
- **color** – color as *AutoCAD Color Index (ACI)* see [add_line\(\)](#)
- **linetype** – line type as string see [add_line\(\)](#)

add_arc (*center*: Sequence[float], *radius*: float, *start*: float = 0, *end*: float = 360, *layer*: str = '0', *color*: int = None, *linetype*: str = None) → None
Add an ARC entity. The arc goes counter clockwise from *start* angle to *end* angle.

Parameters

- **center** – arc center point as (x, y) tuple
- **radius** – arc radius as float
- **start** – arc start angle in degrees as float
- **end** – arc end angle in degrees as float
- **layer** – layer name as string see [add_line\(\)](#)
- **color** – color as *AutoCAD Color Index (ACI)* see [add_line\(\)](#)
- **linetype** – line type as string see [add_line\(\)](#)

add_point (*location*: Sequence[float], *layer*: str = '0', *color*: int = None, *linetype*: str = None) → None
Add a POINT entity.

Parameters

- **location** – point location as (x, y [, z]) tuple
- **layer** – layer name as string see [add_line\(\)](#)
- **color** – color as *AutoCAD Color Index (ACI)* see [add_line\(\)](#)
- **linetype** – line type as string see [add_line\(\)](#)

add_3dface (*vertices*: Iterable[Sequence[float]], *invisible*: int = 0, *layer*: str = '0', *color*: int = None, *linetype*: str = None) → None

Add a 3DFACE entity. 3DFACE is a spatial area with 3 or 4 vertices, all vertices have to be in the same plane.

Parameters

- **vertices** – iterable of 3 or 4 (`x`, `y`, `z`) vertices.
- **invisible** – bit coded flag to define the invisible edges,
 1. edge = 1
 2. edge = 2
 3. edge = 4
 4. edge = 8Add edge values to set multiple edges invisible, 1. edge + 3. edge = 1 + 4 = 5, all edges = 15
- **layer** – layer name as string see [add_line\(\)](#)
- **color** – color as *AutoCAD Color Index (ACI)* see [add_line\(\)](#)
- **linetype** – line type as string see [add_line\(\)](#)

add_solid(`vertices: Iterable[Sequence[float]]`, `layer: str = '0'`, `color: int = None`, `linetype: str = None`) → None

Add a SOLID entity. SOLID is a solid filled area with 3 or 4 edges and SOLID is a 2D entity.

Parameters

- **vertices** – iterable of 3 or 4 (`x`, `y` [, `z`]) tuples, z-axis will be ignored.
- **layer** – layer name as string see [add_line\(\)](#)
- **color** – color as *AutoCAD Color Index (ACI)* see [add_line\(\)](#)
- **linetype** – line type as string see [add_line\(\)](#)

add_polyline_2d(`points: Iterable[Sequence[T_co]]`, `format: str = 'xy'`, `closed: bool = False`, `start_width: float = 0`, `end_width: float = 0`, `layer: str = '0'`, `color: int = None`, `linetype: str = None`) → None

Add a 2D POLYLINE entity with start width, end width and bulge value support.

Format codes:

x	x-coordinate
y	y-coordinate
s	start width
e	end width
b	bulge value
v	(x, y) tuple (z-axis is ignored)

Parameters

- **points** – iterable of (`x`, `y`, [`start_width`, `[end_width`, `[bulge]`]) tuple, value order according to the `format` string, unset values default to 0
- **format** – format string, default is 'xy'
- **closed** – True creates a closed polyline
- **start_width** – default start width, default is 0
- **end_width** – default end width, default is 0
- **layer** – layer name as string see [add_line\(\)](#)
- **color** – color as *AutoCAD Color Index (ACI)* see [add_line\(\)](#)

- **linetype** – line type as string see `add_line()`

add polyline (`vertices: Iterable[Sequence[float]]`, `closed: bool = False`, `layer: str = '0'`, `color: int = None`, `linetype: str = None`) → None
Add a 3D POLYLINE entity.

Parameters

- **vertices** – iterable of (x, y[, z]) tuples, z-axis is 0 by default
- **closed** – True creates a closed polyline
- **layer** – layer name as string see `add_line()`
- **color** – color as *AutoCAD Color Index (ACI)* see `add_line()`
- **linetype** – line type as string see `add_line()`

Changed in version 0.12: Write only 3D POLYLINE entity, added `closed` argument.

add_polyface (`vertices: Iterable[Sequence[float]]`, `faces: Iterable[Sequence[int]]`, `layer: str = '0'`, `color: int = None`, `linetype: str = None`) → None

Add a POLYFACE entity. The POLYFACE entity supports only faces of maximum 4 vertices, more indices will be ignored. A simple square would be:

```
v0 = (0, 0, 0)
v1 = (1, 0, 0)
v2 = (1, 1, 0)
v3 = (0, 1, 0)
dxf.add_polyface(vertices=[v0, v1, v2, v3], faces=[(0, 1, 2, 3)])
```

All 3D form functions of the `ezdxf.render.forms` module return `MeshBuilder` objects, which provide the required vertex and face lists.

See sphere example: <https://github.com/mozman/ezdxf/blob/master/examples/r12writer.py>

Parameters

- **vertices** – iterable of (x, y, z) tuples
- **faces** – iterable of 3 or 4 vertex indices, indices have to be 0-based
- **layer** – layer name as string see `add_line()`
- **color** – color as *AutoCAD Color Index (ACI)* see `add_line()`
- **linetype** – line type as string see `add_line()`

add_polymesh (`vertices: Iterable[Sequence[float]]`, `size: Tuple[int, int]`, `closed=(False, False)`, `layer: str = '0'`, `color: int = None`, `linetype: str = None`) → None

Add a POLYMESH entity. A POLYMESH is a mesh of m rows and n columns, each mesh vertex has its own x-, y- and z coordinates. The mesh can be closed in m- and/or n-direction. The vertices have to be in column order: (m0, n0), (m0, n1), (m0, n2), (m1, n0), (m1, n1), (m1, n2), ...

See example: <https://github.com/mozman/ezdxf/blob/master/examples/r12writer.py>

Parameters

- **vertices** – iterable of (x, y, z) tuples, in column order
- **size** – mesh dimension as (m, n)-tuple, requirement: `len(vertices) == m*n`
- **closed** – (m_closed, n_closed) tuple, for closed mesh in m and/or n direction
- **layer** – layer name as string see `add_line()`
- **color** – color as *AutoCAD Color Index (ACI)* see `add_line()`

- **linetype** – line type as string see [add_line\(\)](#)

add_text (*text: str, insert: Sequence[float] = (0, 0), height: float = 1.0, width: float = 1.0, align: str = 'LEFT', rotation: float = 0.0, oblique: float = 0.0, style: str = 'STANDARD', layer: str = '0', color: int = None*) → None
Add a one line TEXT entity.

Parameters

- **text** – the text as string
- **insert** – insert location as (x, y) tuple
- **height** – text height in drawing units
- **width** – text width as factor
- **align** – text alignment, see table below
- **rotation** – text rotation in degrees as float
- **oblique** – oblique in degrees as float, vertical = 0 (default)
- **style** – text style name as string, if FIXED-TABLES are written some predefined text styles are available, else text style is always 'STANDARD'.
- **layer** – layer name as string see [add_line\(\)](#)
- **color** – color as [AutoCAD Color Index \(ACI\)](#) see [add_line\(\)](#)

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

The special alignments ALIGNED and FIT are not available.

6.11.7 ODA File Converter Support

Use an installed [ODA File Converter](#) for converting between different versions of .dwg, .dxg and .dxf.

Warning: Execution of an external application is a big security issue! Especially when the path to the executable can be altered.

To avoid this problem delete the `ezdxf.addons.odafc.py` module.

The [ODA File Converter](#) has to be installed by the user, the application is available for Windows XP, Windows 7 or later, Mac OS X, and Linux in 32/64-bit RPM and DEB format.

At least at Windows the GUI of the ODA File Converter pops up on every call.

ODA File Converter version strings, you can use any of this strings to specify a version, 'R...' and 'AC....' strings will be automatically mapped to 'ACAD....' strings:

ODAFC	ezdxf	Version
ACAD9	not supported	AC1004
ACAD10	not supported	AC1006
ACAD12	R12	AC1009
ACAD13	R13	AC1012
ACAD14	R14	AC1014
ACAD2000	R2000	AC1015
ACAD2004	R2004	AC1018
ACAD2007	R2007	AC1021
ACAD2010	R2010	AC1024
ACAD2013	R2013	AC1027
ACAD2018	R2018	AC1032

On Windows systems the path of the `ODAFileConverter.exe` application is stored in the config file (see `ezdxf.options`) in the “odafc-addon” section as key “`win_exec_path`”, the default entry is:

```
[odafc-addon]
win_exec_path = "C:\Program Files\ODA\ODAFileConverter\ODAFileConverter.exe"
```

On Linux and macOS the `ODAFileConverter` command is located by the `shutil.which()` function.

Usage:

```
from ezdxf.addons import odafc

# Load a DWG file
doc = odafc.readfile('my.dwg')

# Use loaded document like any other ezdxf document
print(f'Document loaded as DXF version: {doc.dxfversion}.')
msp = doc.modelspace()
...

# Export document as DWG file for AutoCAD R2018
odafc.export_dwg(doc, 'my_R2018.dwg', version='R2018')
```

ezdxf.addons.odafc.`win_exec_path`

Path to installed `ODA File Converter` executable on Windows systems, default is “`C:\Program Files\ODA\ODAFileConverter\ODAFileConverter.exe`”.

ezdxf.addons.odafc.`readfile`(filename: str, version: str = None, audit=False) → Drawing

Use an installed `ODA File Converter` to convert a DWG/DXB/DXF file into a temporary DXF file and load this file by `ezdxf`.

Parameters

- **filename** – file to load by `ODA File Converter`
- **version** – load file as specific DXF version, by default the same version as the source file or if not detectable the latest by `ezdxf` supported version.
- **audit** – audit source file before loading

ezdxf.addons.odafc.`export_dwg`(doc: Drawing, filename: str, version: str = None, audit=False, replace=False) → None

Use an installed `ODA File Converter` to export a DXF document `doc` as a DWG file.

Saves a temporary DXF file and convert this DXF file into a DWG file by the `ODA File Converter`. If `version` is not specified the DXF version of the source document is used.

Parameters

- **doc** – *ezdxf* DXF document as `Drawing` object
- **filename** – export filename of DWG file, extension will be changed to “.dwg”
- **version** – export file as specific version, by default the same version as the source document.
- **audit** – audit source file by ODA File Converter at exporting
- **replace** – replace existing DWG file if `True`

Changed in version 0.15: added `replace` option

6.11.8 `text2path`

New in version 0.16.

Tools to convert text strings and text based DXF entities into outer- and inner linear paths as `Path` objects. These tools depend on the optional `Matplotlib` package. At the moment only the TEXT and the ATTRIB entity can be converted into paths and hatches.

Don’t expect a 100% match compared to CAD applications.

Text Alignments

The text alignments are enums of type `ezdxf.enums.TextEntityAlignment`

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

The vertical middle alignments (MIDDLE_XXX), center the text vertically in the middle of the uppercase letter “X” (cap height).

Special alignments, where the horizontal alignment is always in the center of the text:

- **ALIGNED**: text is scaled to match the given `length`, scales x- and y-direction by the same factor.
- **FIT**: text is scaled to match the given `length`, but scales only in x-direction.
- **MIDDLE**: insertion point is the center of the total height (cap height + descender height) without scaling, the `length` argument is ignored.

Font Face Definition

A font face is defined by the Matplotlib compatible `FontFace` object by `font-family`, `font-style`, `font-stretch` and `font-weight`.

See also:

- [Font Anatomy](#)
- [Font Properties](#)

String Functions

Entity Functions

6.11.9 MTextExplode

This tool is meant to explode MTEXT entities into single line TEXT entities by replicating the MTEXT layout as close as possible. This tool requires the optional Matplotlib package to create usable results, nonetheless it also works without Matplotlib, but then uses a mono-spaced replacement font for text size measuring which leads to very inaccurate results.

The supported MTEXT features are:

- changing text color
- text strokes: underline, overline and strike through
- changing text size, width and oblique
- changing font faces
- stacked text (fractions)
- multi-column support
- background color
- text frame

The tool requires an initialized DXF document to implement all these features by creating additional text styles. When exploding multiple MTEXT entities, they can share this new text styles. Call the `MTextExplode.finalize()` method just once after all MTEXT entities are processed to create the required text styles, or use `MTextExplode` as context manager by using the `with` statement, see examples below.

There are also many limitations:

- A 100% accurate result cannot be achieved.
- Character tracking is not supported.
- Tabulator stops have only limited support for LEFT and JUSTIFIED aligned paragraphs to support numbered and bullet lists. An excessive use of tabs will lead to incorrect results.
- The DISTRIBUTED alignment will be replaced by the JUSTIFIED alignment.
- Text flow is always “left to right”.
- The line spacing mostly corresponds to the “EXACT” style, except for stacked text (fractions), which corresponds more to the “AT LEAST” style, but not precisely. This behavior maybe will improve in the future.
- FIELDS are not evaluated by `ezdxf`.

`class ezdxf.addons.MTextExplode(layout, doc=None, spacing_factor = 1.0)`

The `MTextExplode` class is a tool to disassemble MTEXT entities into single line TEXT entities and additional LINE entities if required to emulate strokes.

The `layout` argument defines the target layout for “exploded” parts of the MTEXT entity. Use argument `doc` if the target layout has no DXF document assigned. The `spacing_factor` argument is an advanced tuning parameter to scale the size of space chars.

`explode(mtext: MText, destroy=True)`

Explode `mtext` and destroy the source entity if argument `destroy` is True.

`finalize()`

Create required text styles. This method is called automatically if the class is used as context manager.

Example to explode all MTEXT entities in the DXF file “mtext.dxf”:

```
import ezdxf
from ezdxf.addons import MTTextExplode

doc = ezdxf.readfile("mtext.dxf")
msp = doc.modelspace()
with MTTextExplode(msp) as xpl:
    for mtext in msp.query("MTEXT"):
        xpl.explode(mtext)
doc.saveas("xpl_mtext.dxf")
```

Explode all MTEXT entities into the block “EXPLODE”:

```
import ezdxf
from ezdxf.addons import MTTextExplode

doc = ezdxf.readfile("mtext.dxf")
msp = doc.modelspace()
blk = doc.blocks.new("EXPLODE")
with MTTextExplode(blk) as xpl:
    for mtext in msp.query("MTEXT"):
        xpl.explode(mtext)
msp.add_block_ref("EXPLODE", (0, 0))
doc.saveas("xpl_into_block.dxf")
```

6.11.10 PyCSG

Constructive Solid Geometry (CSG) is a modeling technique that uses Boolean operations like union and intersection to combine 3D solids. This library implements CSG operations on meshes elegantly and concisely using BSP trees, and is meant to serve as an easily understandable implementation of the algorithm. All edge cases involving overlapping coplanar polygons in both solids are correctly handled.

Example for usage:

```
import ezdxf
from ezdxf.render.forms import cube, cylinder_2p
from ezdxf.addons.pyCSG import CSG

# create new DXF document
doc = ezdxf.new()
msp = doc.modelspace()

# create same geometric primitives as MeshTransformer() objects
cube1 = cube()
cylinder1 = cylinder_2p(count=32, base_center=(0, -1, 0), top_center=(0, 1, 0), radius=.25)

# build solid union
union = CSG(cube1) + CSG(cylinder1)
# convert to mesh and render mesh to modelspace
union.mesh().render(msp, dxftattribs={'color': 1})

# build solid difference
difference = CSG(cube1) - CSG(cylinder1)
# convert to mesh, translate mesh and render mesh to modelspace
```

(continues on next page)

(continued from previous page)

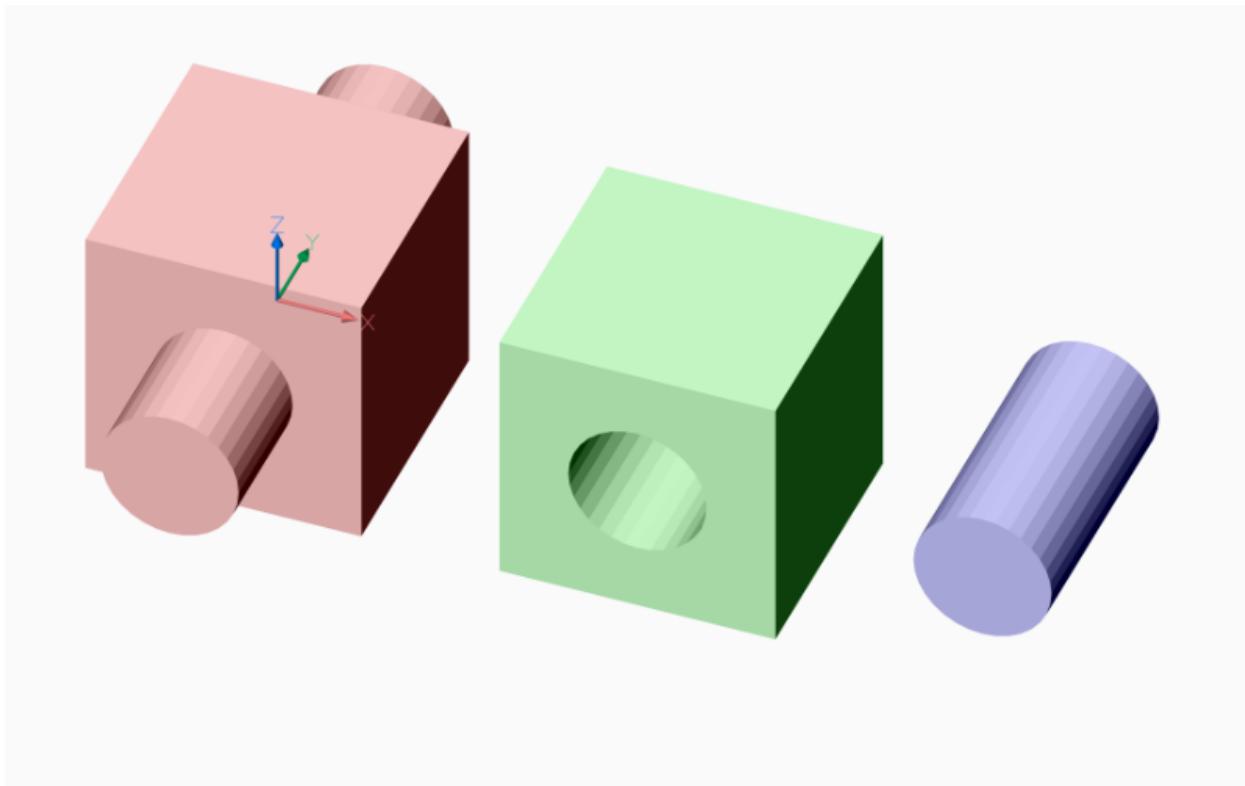
```

difference.mesh().translate(1.5).render(msp, dxffattribs={'color': 3})

# build solid intersection
intersection = CSG(cube1) * CSG(cylinder1)
# convert to mesh, translate mesh and render mesh to modelspace
intersection.mesh().translate(2.75).render(msp, dxffattribs={'color': 5})

doc.saveas('csg.dxf')

```



This CSG kernel supports only meshes as `MeshBuilder` objects, which can be created from and converted to DXF `Mesh` entities.

This CSG kernel is **not** compatible with ACIS objects like `Solid3d`, `Body`, `Surface` or `Region`.

Note: This is a pure Python implementation, don't expect great performance and the implementation is based on an unbalanced `BSP` tree, so in the case of `RecursionError`, increase the recursion limit:

```

import sys

actual_limit = sys.getrecursionlimit()
# default is 1000, increasing too much may cause a seg fault
sys.setrecursionlimit(10000)

... # do the CSG stuff

sys.setrecursionlimit(actual_limit)

```

CSG works also with spheres, but with really bad runtime behavior and most likely `RecursionError` exceptions,

and use `quadrilaterals` as body faces to reduce face count by setting argument `quads` to True.

```
import ezdxf

from ezdxf.render.forms import sphere, cube
from ezdxf.addons.pyccsg import CSG

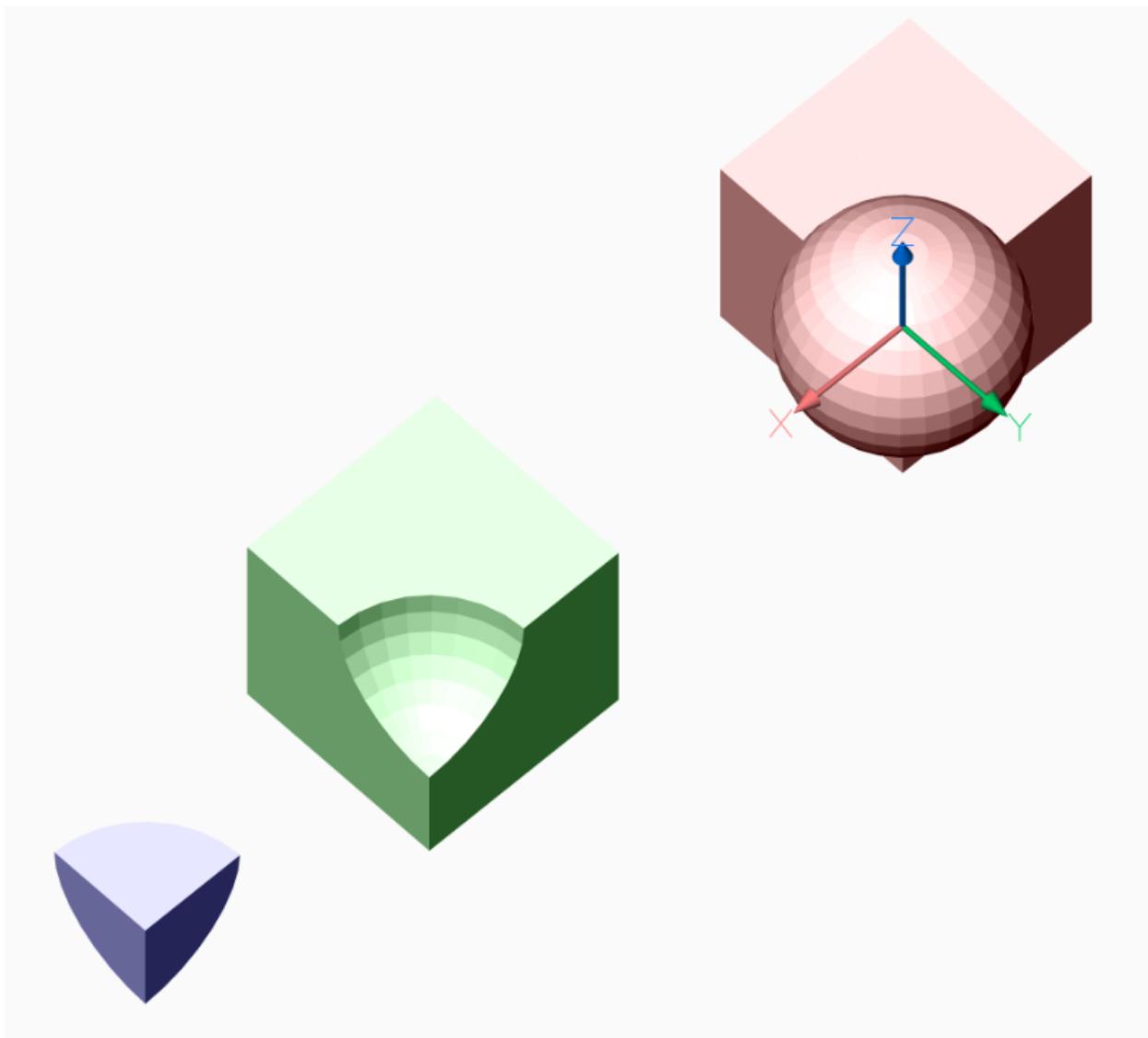
doc = ezdxf.new()
doc.set_modelspace_vport(6, center=(5, 0))
msp = doc.modelspace()

cube1 = cube().translate(-.5, -.5, -.5)
sphere1 = sphere(count=32, stacks=16, radius=.5, quads=True)

union = (CSG(cube1) + CSG(sphere1)).mesh()
union.render(msp, dxftattribs={'color': 1})

subtract = (CSG(cube1) - CSG(sphere1)).mesh().translate(2.5)
subtract.render(msp, dxftattribs={'color': 3})

intersection = (CSG(cube1) * CSG(sphere1)).mesh().translate(4)
intersection.render(msp, dxftattribs={'color': 5})
```



Hard Core CSG - Menger Sponge Level 3 vs Sphere

Required runtime on an old Xeon E5-1620 Workstation @ 3.60GHz, with default recursion limit of 1000 on Windows 10:

- CPython 3.8.1 64bit: ~60 seconds,
- pypy3 [PyPy 7.2.0] 32bit: ~6 seconds, and using `__slots__` reduced runtime below 5 seconds, yes - pypy is worth a look for long running scripts!

```
from ezdxf.render.forms import sphere
from ezdxf.addons import MengerSponge
from ezdxf.addons.pyccsg import CSG

doc = ezdxf.new()
doc.layers.new('sponge', dxfattribs={'color': 5})
doc.layers.new('sphere', dxfattribs={'color': 6})

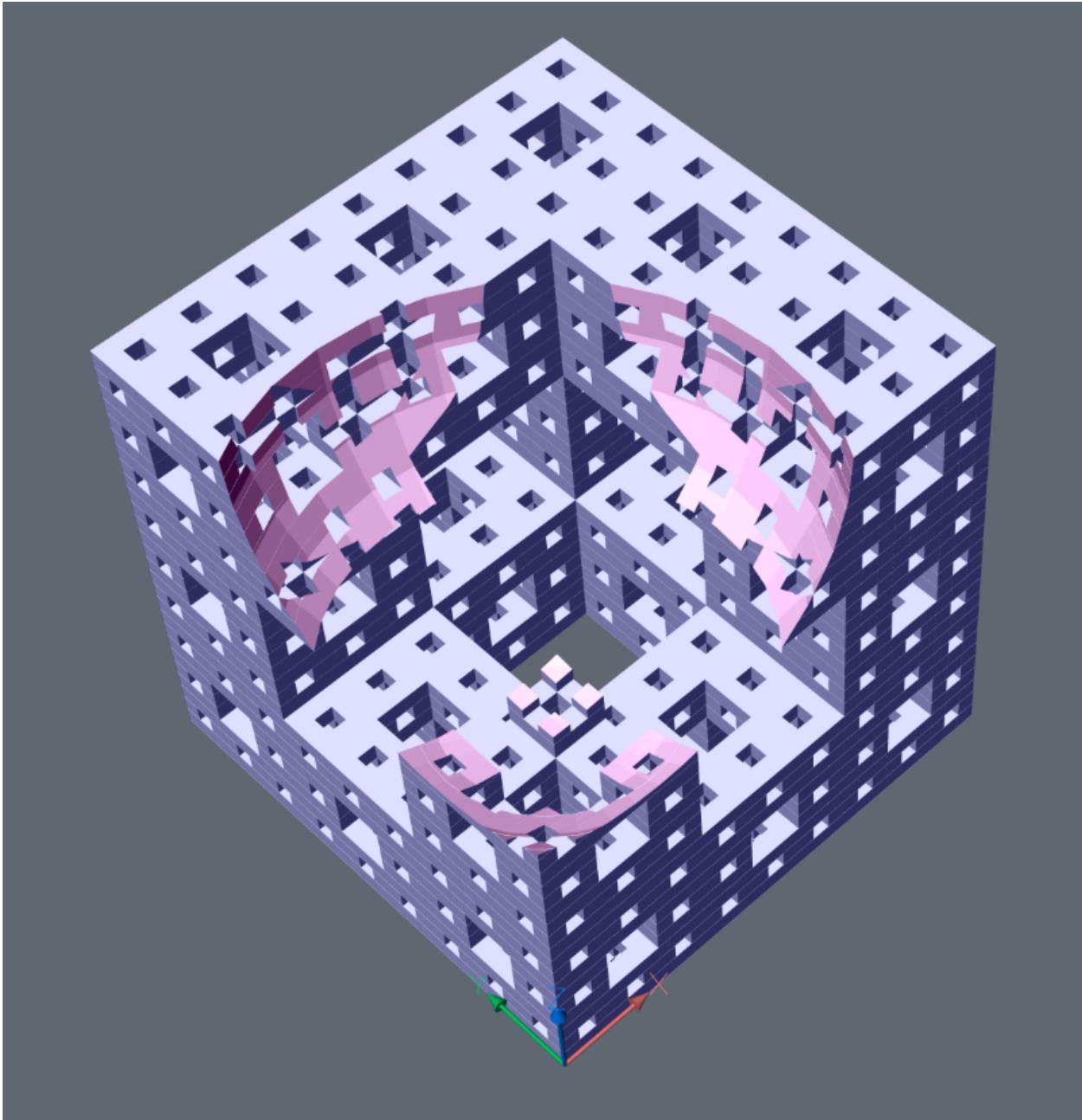
doc.set_modelspace_vport(6, center=(5, 0))
msp = doc.modelspace()
```

(continues on next page)

(continued from previous page)

```
sponge1 = MengerSponge(level=3).mesh()
sphere1 = sphere(count=32, stacks=16, radius=.5, quads=True).translate(.25, .25, 1)

subtract = (CSG(sponge1, meshid=1) - CSG(sphere1, meshid=2))
# get mesh result by id
subtract.mesh(1).render(msp, dxftattribs={'layer': 'sponge'})
subtract.mesh(2).render(msp, dxftattribs={'layer': 'sphere'})
```



CSG Class

class `ezdxf.addons.pycsg.CSG(mesh: MeshBuilder, meshid: int = 0)`

Constructive Solid Geometry (CSG) is a modeling technique that uses Boolean operations like union and intersection to combine 3D solids. This class implements CSG operations on meshes.

New 3D solids are created from `MeshBuilder` objects and results can be exported as `MeshTransformer` objects to `ezdxf` by method `mesh()`.

Parameters

- `mesh` – `ezdxf.render.MeshBuilder` or inherited object
- `meshid` – individual mesh ID to separate result meshes, 0 is default

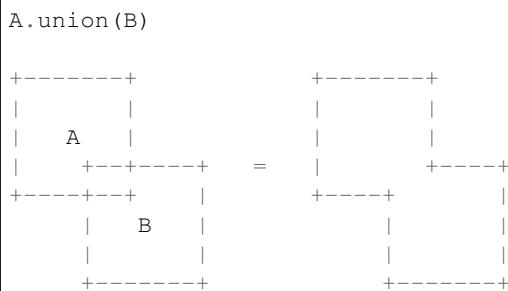
mesh(meshid: int = 0) → MeshTransformer

Returns a `ezdxf.render.MeshTransformer` object.

Parameters `meshid` – individual mesh ID, 0 is default

union(other: CSG) → CSG

Return a new CSG solid representing space in either this solid or in the solid `other`. Neither this solid nor the solid `other` are modified:

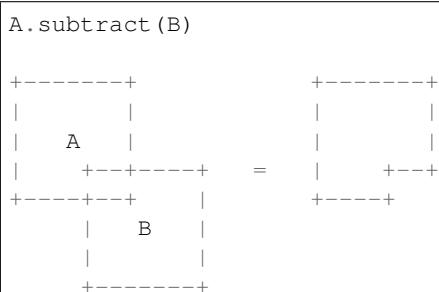


__add__(other: CSG) → CSG

```
union = A + B
```

subtract(other: CSG) → CSG

Return a new CSG solid representing space in this solid but not in the solid `other`. Neither this solid nor the solid `other` are modified:



__sub__(other: CSG) → CSG

```
difference = A - B
```

intersect (*other: CSG*) → CSG

Return a new CSG solid representing space both this solid and in the solid *other*. Neither this solid nor the solid *other* are modified:

```
A.intersect(B)

+-----+
|       |
|   A   |
| +---+---+ = +---+
+---+---+ | +---+
|   B   |
|       |
+-----+
```

__mul__ (*other: CSG*) → CSG

```
intersection = A * B
```

inverse() → CSG

Return a new CSG solid with solid and empty space switched. This solid is not modified.

License

- Original implementation [csg.js](#), Copyright (c) 2011 Evan Wallace (<http://madebyevan.com/>), under the MIT license.
- Python port [pycsg](#), Copyright (c) 2012 Tim Knip (<http://www.floorplanner.com>), under the MIT license.
- Additions by Alex Pletzer (Pennsylvania State University)
- Integration as *ezdxf* add-on, Copyright (c) 2020, Manfred Moitzi, MIT License.

6.11.11 Plot Style Files (CTB/STB)

CTB and STB files store plot styles used by AutoCAD and BricsCAD for printing and plotting.

If the plot style table is attached to a *Paperspace* or the *Modelspace*, a change of a plot style affects any object that uses that plot style. CTB files contain color dependent plot style tables, STB files contain named plot style tables.

See also:

- Using plot style tables in AutoCAD
- AutoCAD Plot Style Table Editor
- BricsCAD Plot Style Table Editor
- AUTODESK KNOWLEDGE NETWORK: How to [install](#) CTB files in AutoCAD

`ezdxf.addons.acadctb.load(filename: str) → Union[ColorDependentPlotStyles, NamedPlotStyles]`

Load the CTB or STB file *filename* from file system.

`ezdxf.addons.acadctb.new_ctb() → ColorDependentPlotStyles`

Create a new CTB file.

Changed in version 0.10: renamed from `new()`

`ezdxf.addons.acadctb.new_stb()` → NamedPlotStyles
Create a new STB file.

ColorDependentPlotStyles

Color dependent plot style table (CTB file), table entries are *PlotStyle* objects.

class `ezdxf.addons.acadctb.ColorDependentPlotStyles`

description

Custom description of plot style file.

scale_factor

Specifies the factor by which to scale non-ISO linetypes and fill patterns.

apply_factor

Specifies whether or not you want to apply the *scale_factor*.

custom_lineweight_display_units

Set 1 for showing linewidth in inch in AutoCAD CTB editor window, but linewidths are always defined in millimeters.

lineweights

Lineweights table as `array.array`

__getitem__(aci: int) → PlotStyle

Returns *PlotStyle* for *AutoCAD Color Index (ACI)* *aci*.

__iter__() → Iterable[PlotStyle]

Iterable of all plot styles.

new_style(aci: int, data: dict = None) → PlotStyle

Set *aci* to new attributes defined by *data* dict.

Parameters

- **aci** – *AutoCAD Color Index (ACI)*
- **data** – dict of *PlotStyle* attributes: description, color, physical_pen_number, virtual_pen_number, screen, linepattern_size, linetype, adaptive_linetype, linewidth, end_style, join_style, fill_style

get_lineweight(aci: int)

Returns the assigned linewidth for *PlotStyle* *aci* in millimeter.

get_lineweight_index(linewidth: float) → int

Get index of *linewidth* in the linewidth table or append *linewidth* to linewidth table.

get_table_lineweight(index: int) → float

Returns linewidth in millimeters of linewidth table entry *index*.

Parameters **index** – linewidth table index = *PlotStyle.linewidth*

Returns linewidth in mm or 0.0 for use entity linewidth

set_table_lineweight(index: int, linewidth: float) → int

Argument *index* is the linewidth table index, not the *AutoCAD Color Index (ACI)*.

Parameters

- **index** – linewidth table index = *PlotStyle.linewidth*
- **linewidth** – in millimeters

save (*filename: str*) → None
Save CTB file as *filename* to the file system.

write (*stream: BinaryIO*) → None
Compress and write CTB file to binary *stream*.

NamedPlotStyles

Named plot style table (STB file), table entries are *PlotStyle* objects.

class ezdxf.addons.acadctb.**NamedPlotStyles**

description
Custom description of plot style file.

scale_factor
Specifies the factor by which to scale non-ISO linetypes and fill patterns.

apply_factor
Specifies whether or not you want to apply the *scale_factor*.

custom_lineweight_display_units
Set 1 for showing linewidth in inch in AutoCAD CTB editor window, but linewidths are always defined in millimeters.

lineweights
Lineweights table as `array.array`

__getitem__ (*name: str*) → *PlotStyle*
Returns *PlotStyle* by *name*.

__delitem__ (*name: str*)
Delete plot style *name*. Plot style 'Normal' is not deletable.

__iter__ () → `Iterable[str]`
Iterable of all plot style names.

new_style (*name: str, localized_name: str = None, data: dict = None*) → *PlotStyle*
Create new class:*PlotStyle* *name* by attribute dict *data*, replaces existing class:*PlotStyle* objects.

Parameters

- **name** – plot style name
- **localized_name** – name shown in plot style editor, uses *name* if None
- **data** – dict of *PlotStyle* attributes: description, color, physical_pen_number, virtual_pen_number, screen, linepattern_size, linetype, adaptive_linetype, linewidth, end_style, join_style, fill_style

get_lineweight (*name: str*)
Returns the assigned linewidth for *PlotStyle* *name* in millimeter.

get_lineweight_index (*lineweight: float*) → int
Get index of *lineweight* in the linewidth table or append *lineweight* to linewidth table.

get_table_lineweight (*index: int*) → float
Returns linewidth in millimeters of linewidth table entry *index*.

Parameters **index** – linewidth table index = *PlotStyle.linewidth*

Returns linewidth in mm or 0.0 for use entity linewidth

set_table_lineweight (*index: int, linewidth: float*) → int
 Argument *index* is the linewidth table index, not the *AutoCAD Color Index (ACI)*.

Parameters

- **index** – linewidth table index = *PlotStyle.linewidth*
- **linewidth** – in millimeters

save (*filename: str*) → None
 Save STB file as *filename* to the file system.

write (*stream: BinaryIO*) → None
 Compress and write STB file to binary *stream*.

PlotStyle

```
class ezdxf.addons.acadctb.PlotStyle
```

index

Table index (0-based). (int)

aci

AutoCAD Color Index (ACI) in range from 1 to 255. Has no meaning for named plot styles. (int)

description

Custom description of plot style. (str)

physical_pen_number

Specifies physical plotter pen, valid range from 1 to 32 or *AUTOMATIC*. (int)

virtual_pen_number

Only used by non-pen plotters and only if they are configured for virtual pens. valid range from 1 to 255 or *AUTOMATIC*. (int)

screen

Specifies the color intensity of the plot on the paper, valid range is from 0 to 100. (int)

If you select 100 the drawing will plotted with its full color intensity. In order for screening to work, the *dithering* option must be active.

linetype

Overrides the entity linetype, default value is *OBJECT_LINETYPE*. (bool)

adaptive_linetype

True if a complete linetype pattern is more important than a correct linetype scaling, default is True. (bool)

linepattern_size

Line pattern size, default = 0.5. (float)

lineweight

Overrides the entity lineWEIGHT, default value is *OBJECT_LINEWEIGHT*. This is an index into the *UserStyles.lineweights* table. (int)

end_style

Line end cap style, see table below, default is END_STYLE_OBJECT (int)

join_style

Line join style, see table below, default is JOIN_STYLE_OBJECT (int)

fill_style

Line fill style, see table below, default is FILL_STYLE_OBJECT (int)

dithering

Depending on the capabilities of your plotter, dithering approximates the colors with dot patterns. When this option is False, the colors are mapped to the nearest color, resulting in a smaller range of colors when plotting.

Dithering is available only whether you select the object's color or assign a plot style color.

grayscale

Plot colors in grayscale. (bool)

Default Line Weights

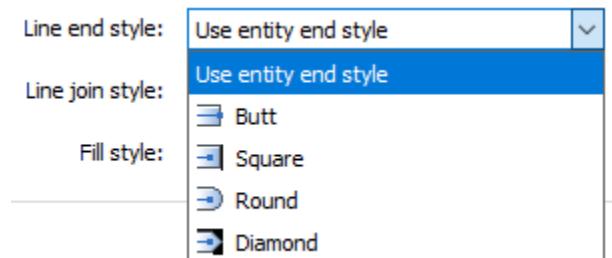
#	[mm]
0	0.00
1	0.05
2	0.09
3	0.10
4	0.13
5	0.15
6	0.18
7	0.20
8	0.25
9	0.30
10	0.35
11	0.40
12	0.45
13	0.50
14	0.53
15	0.60
16	0.65
17	0.70
18	0.80
19	0.90
20	1.00
21	1.06
22	1.20
23	1.40
24	1.58
25	2.00
26	2.11

Predefined Values

```
ezdxf.addons.acadctb.AUTOMATIC
ezdxf.addons.acadctb.OBJECT_LINEWEIGHT
ezdxf.addons.acadctb.OBJECT_LINETYPE
ezdxf.addons.acadctb.OBJECT_COLOR
```

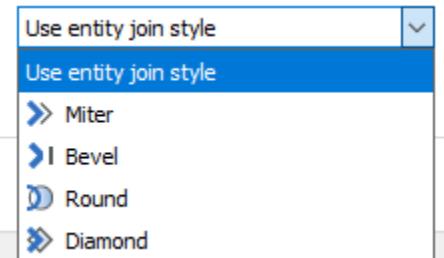
```
ezdxf.addons.acadctb.OBJECT_COLOR2
```

Line End Style



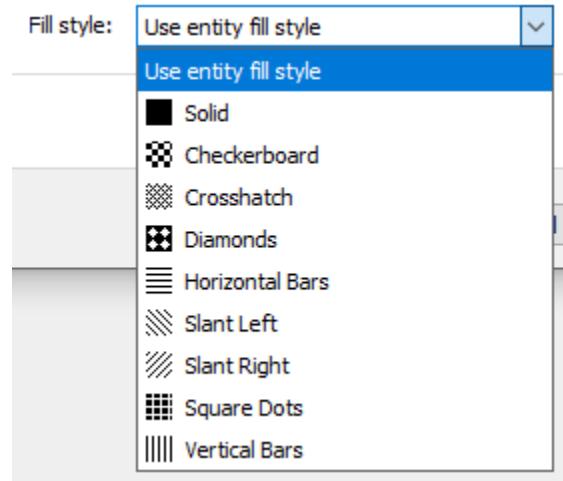
END_STYLE_BUTT	0
END_STYLE_SQUARE	1
END_STYLE_ROUND	2
END_STYLE_DIAMOND	3
END_STYLE_OBJECT	4

Line Join Style



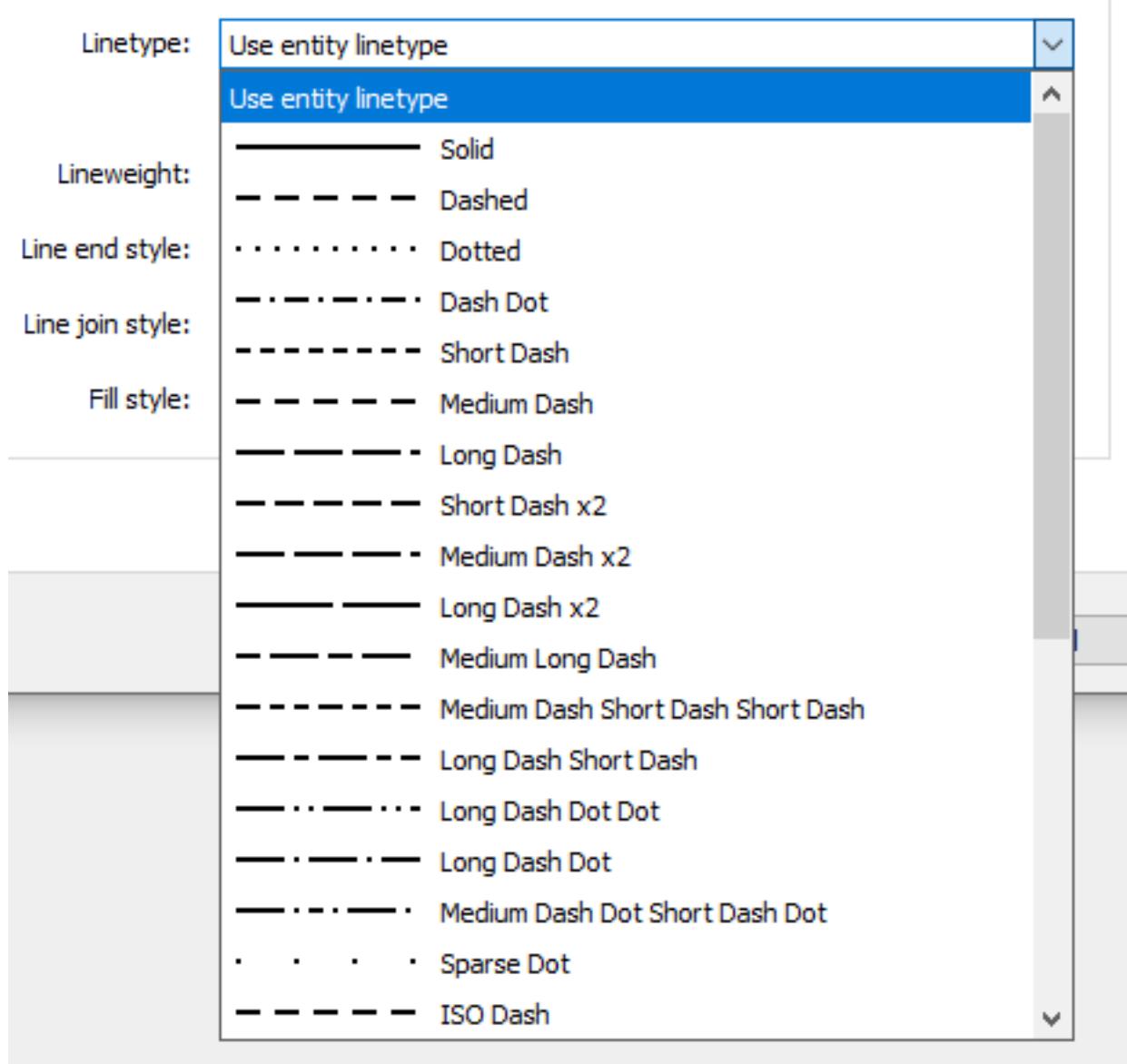
JOIN_STYLE_MITER	0
JOIN_STYLE_BEVEL	1
JOIN_STYLE_ROUND	2
JOIN_STYLE_DIAMOND	3
JOIN_STYLE_OBJECT	5

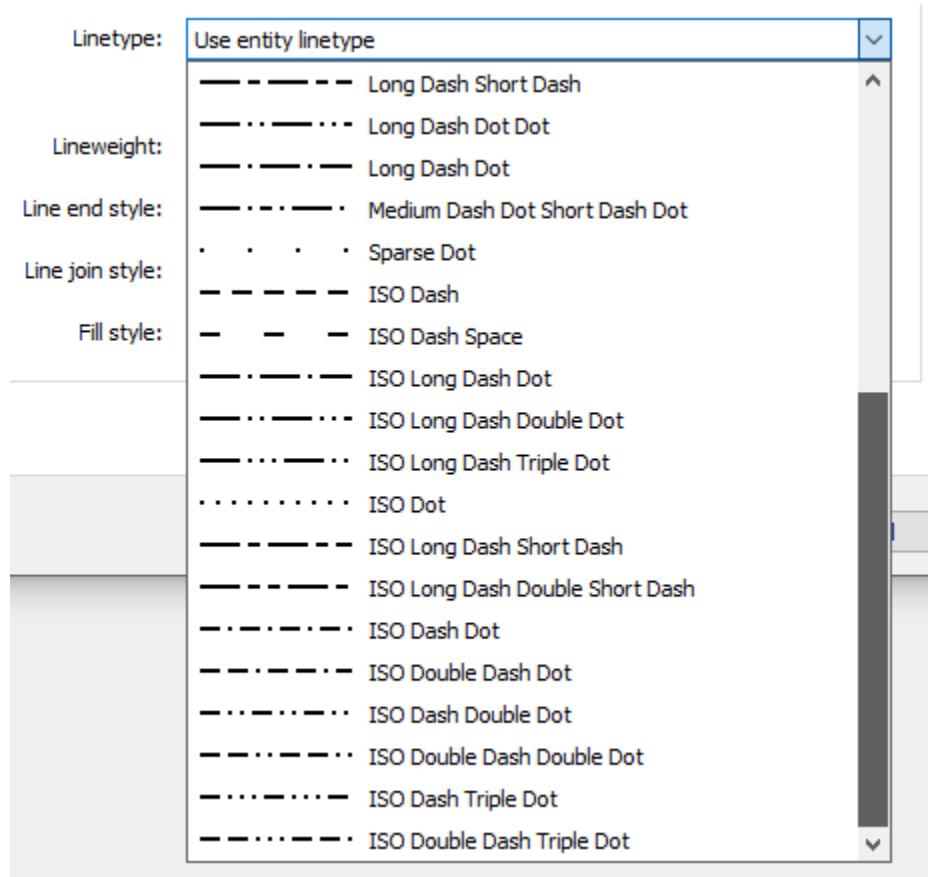
Fill Style



FILL_STYLE_SOLID	64
FILL_STYLE_CHECKERBOARD	65
FILL_STYLE_CROSSHATCH	66
FILL_STYLE_DIAMONDS	67
FILL_STYLE_HORIZONTAL_BARS	68
FILL_STYLE_SLANT_LEFT	69
FILL_STYLE_SLANT_RIGHT	70
FILL_STYLE_SQUARE_DOTS	71
FILL_STYLE_VERICAL_BARS	72
FILL_STYLE_OBJECT	73

Linetypes





Linetype name	Value
Solid	0
Dashed	1
Dotted	2
Dash Dot	3
Short Dash	4
Medium Dash	5
Long Dash	6
Short Dash x2	7
Medium Dash x2	8
Long Dash x2	9
Medium Lang Dash	10
Medium Dash Short Dash Short Dash	11
Long Dash Short Dash	12
Long Dash Dot Dot	13
Long Dash Dot	14
Medium Dash Dot Short Dash Dot	15
Sparse Dot	16
ISO Dash	17
ISO Dash Space	18
ISO Long Dash Dot	19
ISO Long Dash Double Dot	20
ISO Long Dash Triple Dot	21

Continued on next page

Table 1 – continued from previous page

Linetype name	Value
ISO Dot	22
ISO Long Dash Short Dash	23
ISO Long Dash Double Short Dash	24
ISO Dash Dot	25
ISO Double Dash Dot	26
ISO Dash Double Dot	27
ISO Double Dash Double Dot	28
ISO Dash Triple Dot	29
ISO Double Dash Triple Dot	30
Use entity linetype	31

6.11.12 Showcase Forms

MengerSponge

Build a 3D Menger sponge.

```
class ezdxf.addons.MengerSponge (location: Vertex = (0.0, 0.0, 0.0), length: float = 1.0, level: int = 1, kind: int = 0)
```

Parameters

- **location** – location of lower left corner as (x, y, z) tuple
- **length** – side length
- **level** – subdivide level
- **kind** – type of menger sponge

0	Original Menger Sponge
1	Variant XOX
2	Variant OXO
3	Jerusalem Cube

```
render (layout: GenericLayoutType, merge: bool = False, dxfattribs=None, matrix: Matrix44 = None, ucs: UCS = None) → None
```

Renders the menger sponge into layout, set *merge* to True for rendering the whole menger sponge into one MESH entity, set *merge* to False for rendering the individual cubes of the menger sponge as MESH entities.

Parameters

- **layout** – DXF target layout
- **merge** – True for one MESH entity, False for individual MESH entities per cube
- **dfattribs** – DXF attributes for the MESH entities
- **matrix** – apply transformation matrix at rendering
- **ucs** – apply UCS transformation at rendering

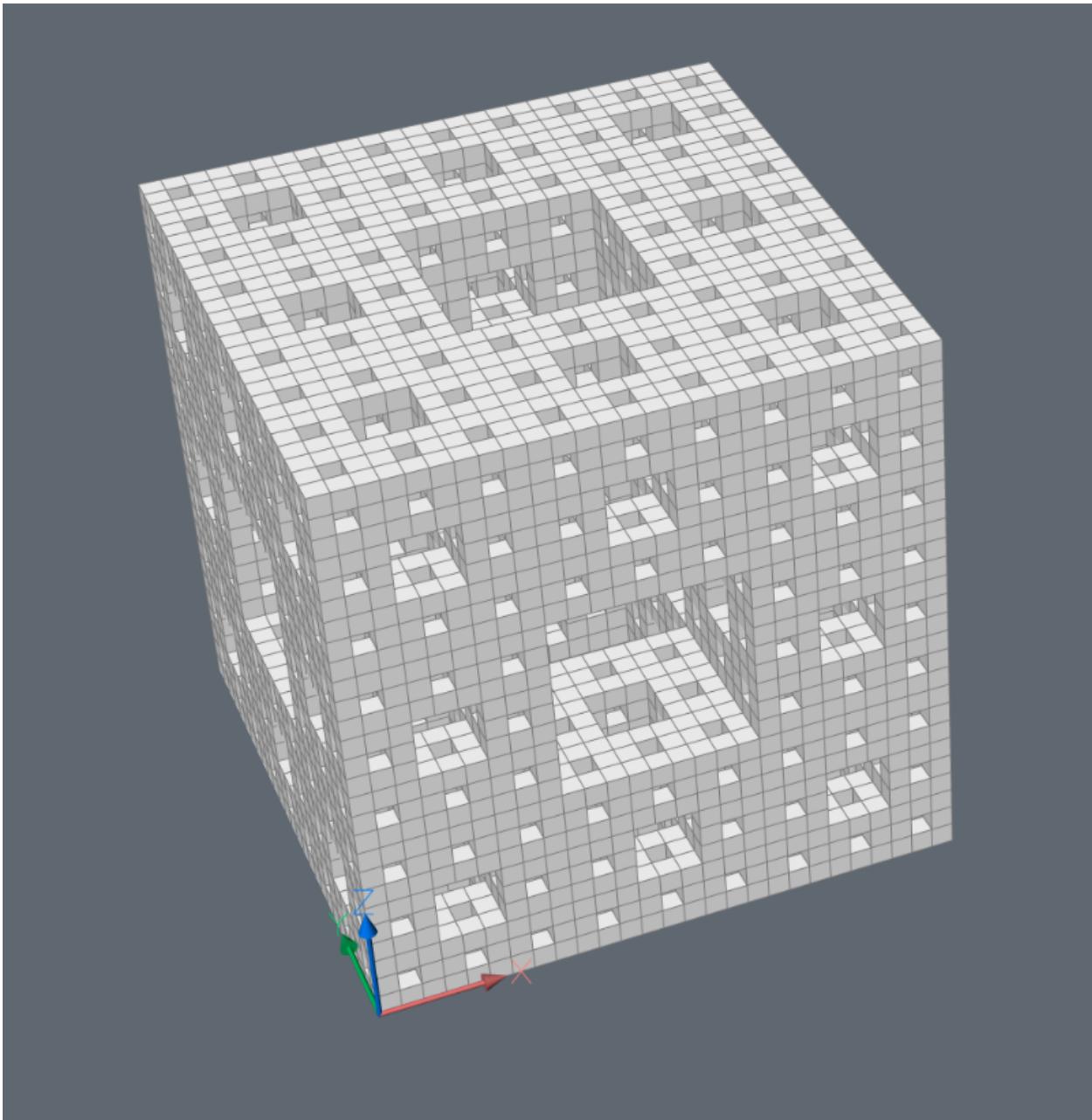
```
cubes () → Iterator[ezdxf.render.mesh.MeshTransformer]
```

Yields all cubes of the menger sponge as individual `MeshTransformer` objects.

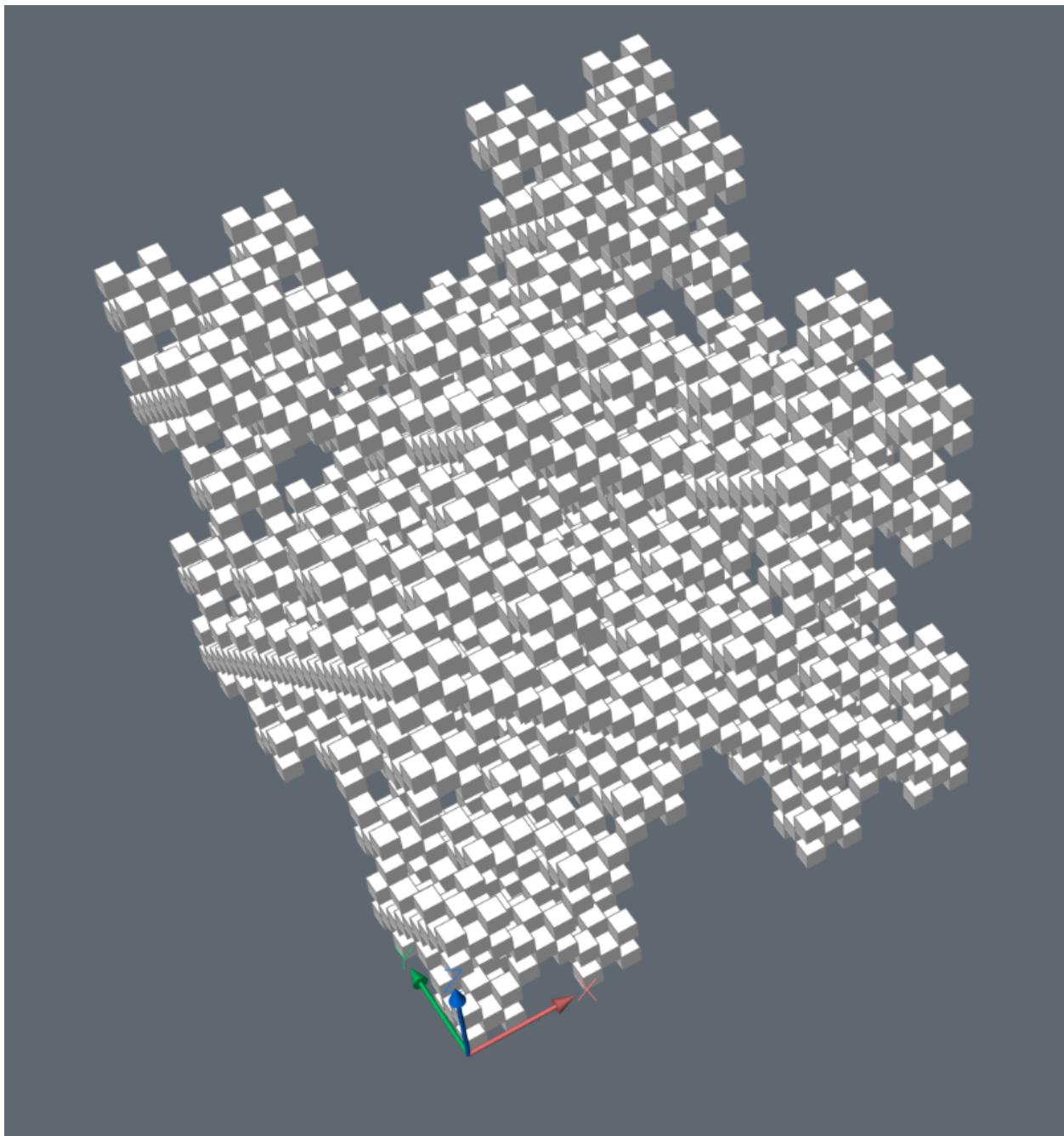
mesh () → ezdxf.render.mesh.MeshTransformer

Returns geometry as one MeshTransformer object.

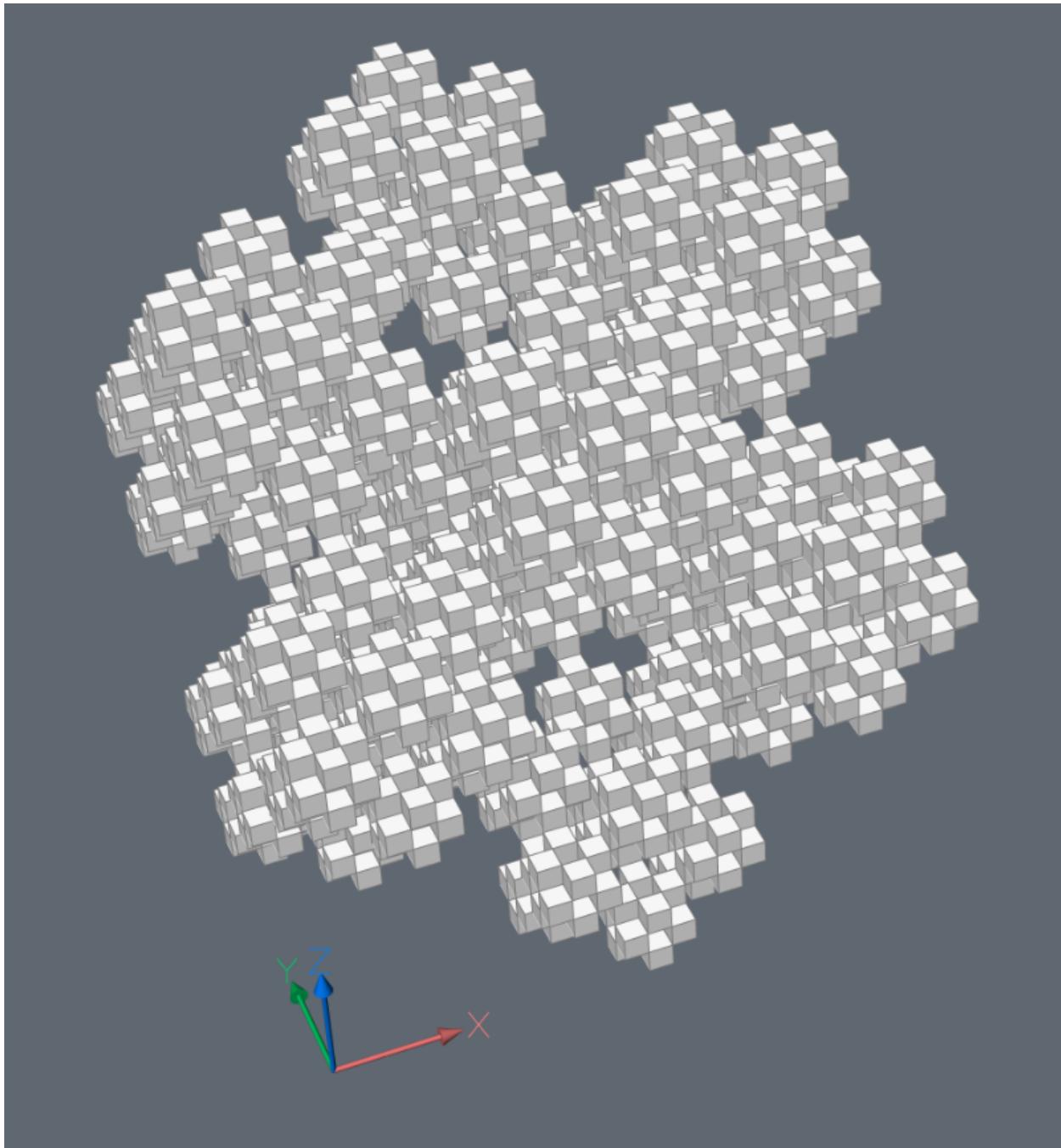
Menger Sponge kind=0:



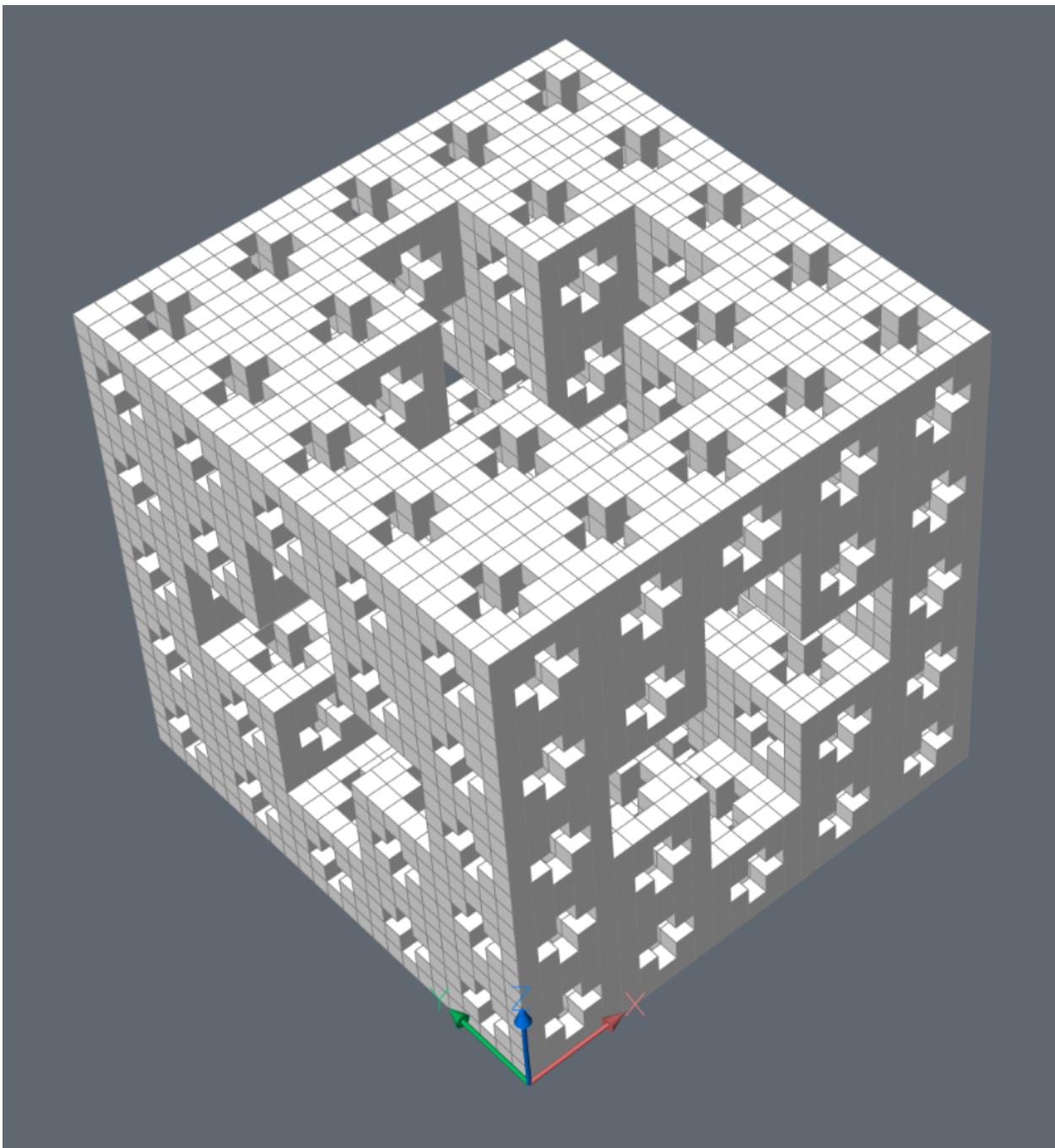
Menger Sponge kind=1:



Menger Sponge kind=2:



Jerusalem Cube kind=3:



SierpinskyPyramid

Build a 3D Sierpinsky Pyramid.

```
class ezdxf.addons.SierpinskyPyramid(location: Vertex = (0.0, 0.0, 0.0), length: float = 1.0,  
level: int = 1, sides: int = 4)
```

Parameters

- **location** – location of base center as (x, y, z) tuple
- **length** – side length

- **level** – subdivide level
- **sides** – sides of base geometry

render (*layout: GenericLayoutType, merge: bool = False, dxfattribs=None, matrix: Matrix44 = None, ucs: UCS = None*) → None

Renders the sierpinsky pyramid into layout, set *merge* to True for rendering the whole sierpinsky pyramid into one MESH entity, set *merge* to False for individual pyramids as MESH entities.

Parameters

- **layout** – DXF target layout
- **merge** – True for one MESH entity, False for individual MESH entities per pyramid
- **dfxfattribs** – DXF attributes for the MESH entities
- **matrix** – apply transformation matrix at rendering
- **ucs** – apply UCS at rendering

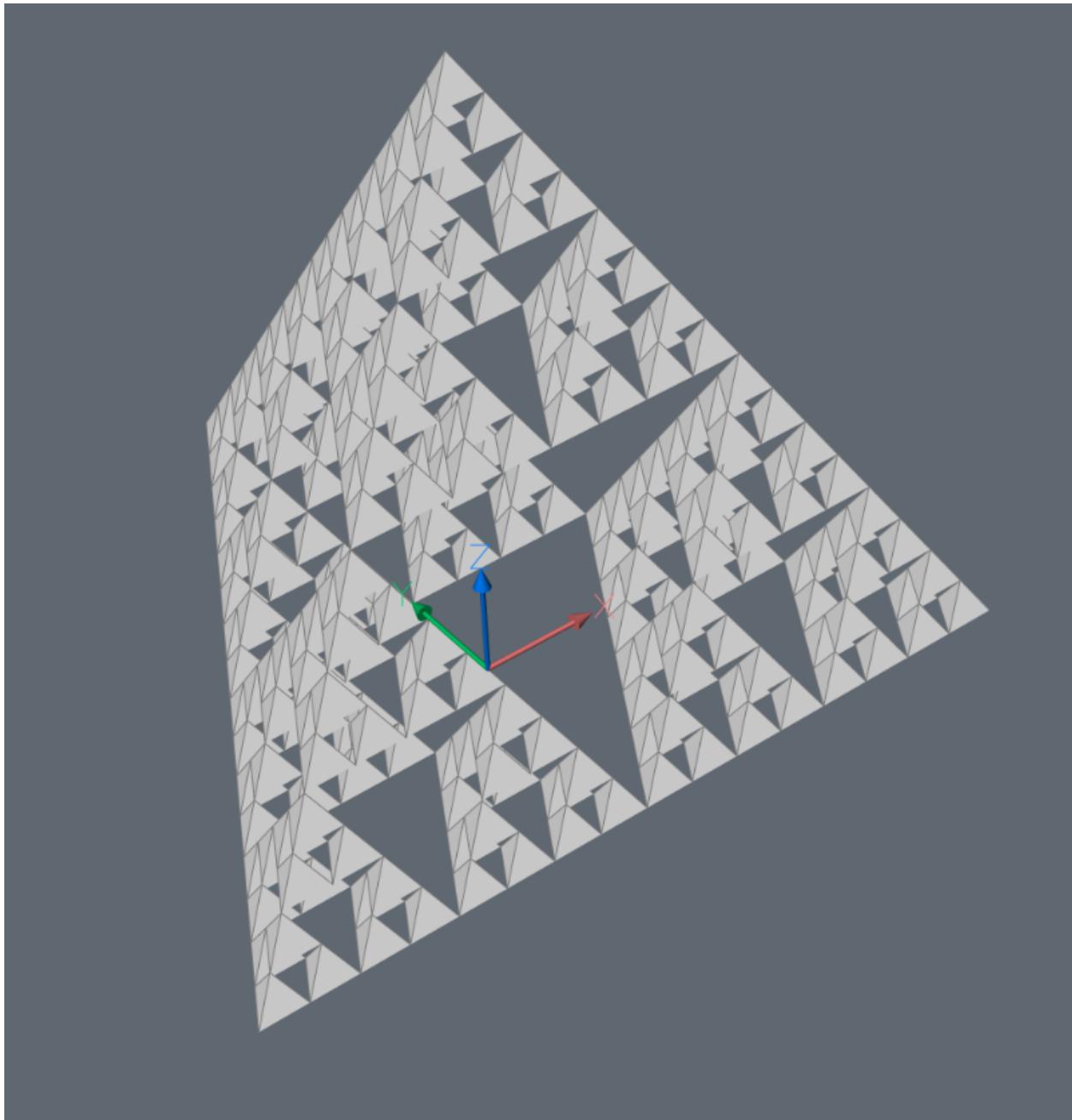
pyramids() → Iterable[ezdxf.render.mesh.MeshTransformer]

Yields all pyramids of the sierpinsky pyramid as individual MeshTransformer objects.

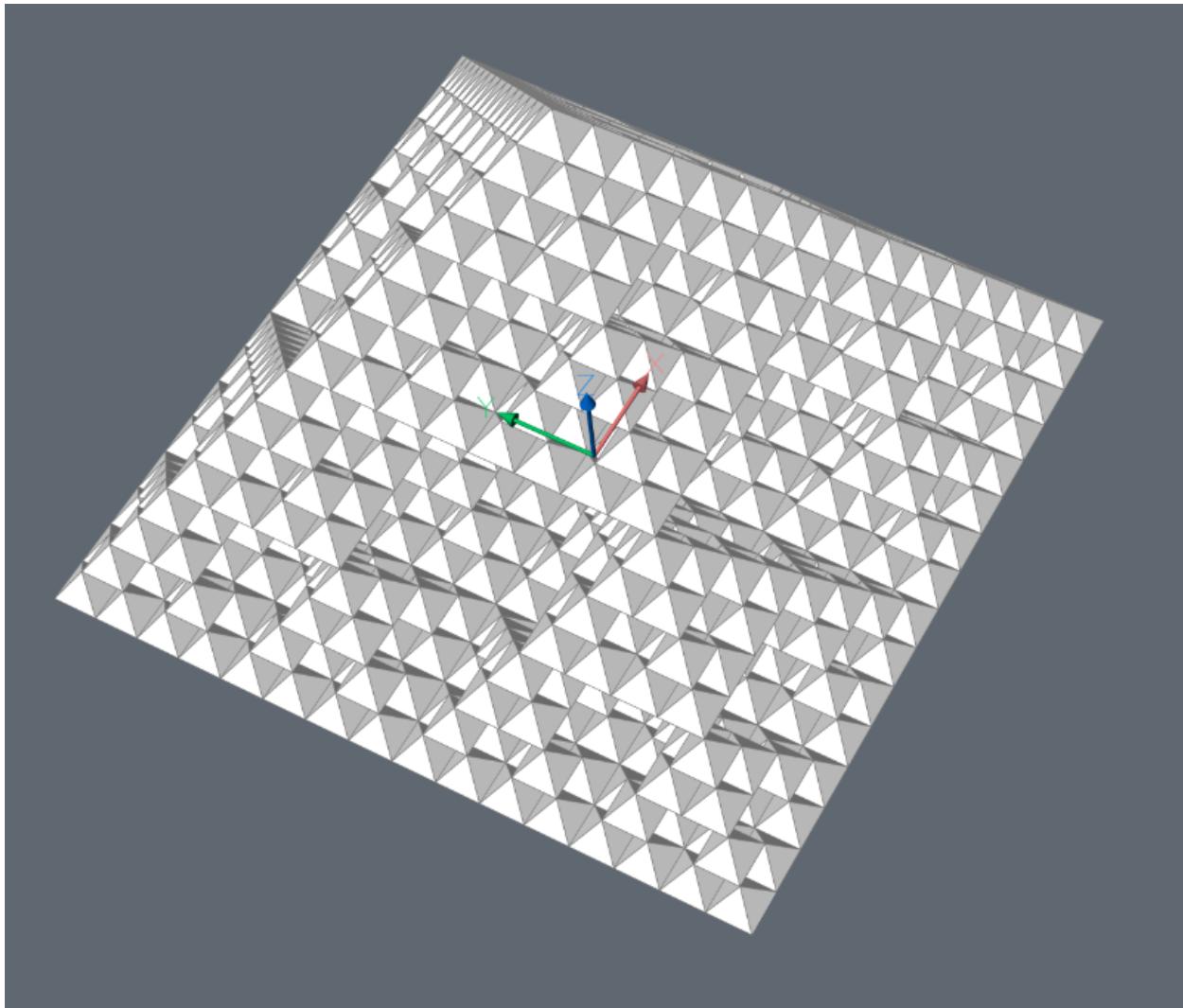
mesh() → ezdxf.render.mesh.MeshTransformer

Returns geometry as one MeshTransformer object.

Sierpinsky Pyramid with triangle base:



Sierpinsky Pyramid with square base:



6.12 DXF Internals

- [DXF Reference](#) provided by Autodesk.
- [DXF Developer Documentation](#) provided by Autodesk.

6.12.1 Basic DXF Structures

DXF File Encoding

DXF R2004 and prior

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

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

Known \$DWGCODEPAGE encodings

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

DXF R2007 and later

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

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

See also:

String value encoding

DXF Tags

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

With the introduction of extended symbol names in DXF R2000, the 255-character limit for strings has been increased to 2049 single-byte characters not including the newline at the end of the line. Nonetheless its safer to use only strings with 255 and less characters, because its not clear if this fact is true for ALL string group codes or only for symbols like layer- or text style names and not all 3rd party libraries may handle this fact correct. The MTEXT content and binary data is still divided into chunks with less than 255 characters.

Group codes are indicating the value type:

Group Code	Value Type
0-9	String
10-39	Double precision 3D point value
40-59	Double-precision floating-point value
60-79	16-bit integer value
90-99	32-bit integer value
100	String
102	String

Continued on next page

Table 2 – continued from previous page

Group Code	Value Type
105	String representing hexadecimal (hex) handle value
110-119	Double precision floating-point value
120-129	Double precision floating-point value
130-139	Double precision floating-point value
140-149	Double precision scalar floating-point value
160-169	64-bit integer value
170-179	16-bit integer value
210-239	Double-precision floating-point value
270-279	16-bit integer value
280-289	16-bit integer value
290-299	Boolean flag value
300-309	Arbitrary text string
310-319	String representing hex value of binary chunk
320-329	String representing hex handle value
330-369	String representing hex object IDs
370-379	16-bit integer value
380-389	16-bit integer value
390-399	String representing hex handle value
400-409	16-bit integer value
410-419	String
420-429	32-bit integer value
430-439	String
440-449	32-bit integer value
450-459	Long
460-469	Double-precision floating-point value
470-479	String
480-481	String representing hex handle value
999	Comment (string)
1000-1009	String
1010-1059	Double-precision floating-point value
1060-1070	16-bit integer value
1071	32-bit integer value

Explanation for some important group codes:

Group Code	Meaning
0	DXF structure tag, entity start/end or table entries
1	The primary text value for an entity
2	A name: Attribute tag, Block name, and so on. Also used to identify a DXF section or table name.
3-4	Other textual or name values
5	Entity handle as hex string (fixed)
6	Line type name (fixed)
7	Text style name (fixed)
8	Layer name (fixed)
9	Variable name identifier (used only in HEADER section of the DXF file)
10	Primary X coordinate (start point of a Line or Text entity, center of a Circle, etc.)
11-18	Other X coordinates
20	Primary Y coordinate. 2n values always correspond to 1n values and immediately follow them in the file (expected by ezdxf)
21-28	Other Y coordinates

Table 3 – continued from previous page

Group Code	Meaning
30	Primary Z coordinate. 3n values always correspond to 1n and 2n values and immediately follow them in the file (except for 31-38)
31-38	Other Z coordinates
39	This entity's thickness if nonzero (fixed)
40-48	Float values (text height, scale factors, etc.)
49	Repeated value - multiple 49 groups may appear in one entity for variable length tables (such as the dash lengths in the LINE entity)
50-58	Angles in degree
62	Color number (fixed)
66	“Entities follow” flag (fixed), only in INSERT and POLYLINE entities
67	Identifies whether entity is in modelspace (0) or paperspace (1)
68	Identifies whether viewport is on but fully off screen, is not active, or is off
69	Viewport identification number
70-78	Integer values such as repeat counts, flag bits, or modes
210, 220, 230	X, Y, and Z components of extrusion direction (fixed)
310	Proxy entity graphics as binary encoded data
330	Owner handle as hex string
347	MATERIAL handle as hex string
348	VISUALSTYLE handle as hex string
370	Lineweight in mm times 100 (e.g. 0.13mm = 13).
390	PLOTSTYLE handle as hex string
420	True color value as 0x00RRGGBB 24-bit value
430	Color name as string
440	Transparency value 0x020000TT 0 = fully transparent / 255 = opaque
999	Comments

For explanation of all group codes see: [DXF Group Codes in Numerical Order Reference](#) provided by Autodesk

Extended Data

DXF R2018 Reference

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

But extended group codes (>=1000) can appear **before** the XDATA section, an example is the BLOCKBASEPOINT-PARAMETER entity in AutoCAD Civil 3D or AutoCAD Map 3D.

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

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

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

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

String value encoding

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

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

String values containing Unicode characters are represented with control character sequences \U+nnnn. (e.g. r'TEST\U+7F3A\U+4E4F\U+89E3\U+91CA\U+6B63THIS\U+56FE')

To support the DXF unicode encoding ezdxfs registers an encoding codec `dxf_backslash_replace`, defined in `ezdxfs.1ldxf.encoding()`.

String values can be stored with these dxf group codes:

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

Multi tag text (MTEXT)

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

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

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

See also:

[DXF File Encoding](#)

DXF R13 and later tag structure

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

Subclass Markers

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

Quote about group codes from the DXF reference

Some group codes that define an entity always appear; others are optional and appear only if their values differ from the defaults.

Do not write programs that **rely on the order given here**. The end of an entity is indicated by the next 0 group, which begins the next entity or indicates the end of the section.

Note: Accommodating DXF files from future releases of AutoCAD will be easier if you write your DXF processing program in a table-driven way, ignore undefined group codes, and make no assumptions about

the order of group codes in an entity. With each new AutoCAD release, new group codes will be added to entities to accommodate additional features.

Usage of group codes in subclasses twice

Some later entities contains the same group code twice for different purposes, so order in the sense of which one comes first is important. (e.g. ATTDEF group code 280)

Tag order is sometimes important especially for AutoCAD

In LWPOLYLINE the order of tags is important, if the *count* tag is not the first tag in the AcDbPolyline subclass, AutoCAD will not close the polyline when the *close* flag is set, by the way other applications like BricsCAD ignores the tag order and renders the polyline always correct.

Extension Dictionary

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

The extension dictionary tag sequence:

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

Persistent Reactors

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

The persistent reactors tag sequence:

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

Application-Defined Codes

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

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

(102, 'YOURAPPID}') is also a valid closing tag:

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

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

Exception XRECORD: Tags with group code 102 and a value string without a preceding "{" or the scheme "YOURAPPID}", should be treated as usual group codes.

Embedded Objects

The concept of embedded objects was introduced with AutoCAD 2018 (DXF version AC1032) and this is the only information I found about it at the Autodesk knowledge base: [Embedded and Encapsulated Objects](#)

Quote from [Embedded and Encapsulated Objects](#):

For DXF filing, the embedded object must be filed out and in after all the data of the encapsulating object has been filed out and in.

A separator is needed between the encapsulating object's data and the subsequent embedded object's data. The separator must be similar in function to the group 0 or 100 in that it must cause the filer to stop reading data. The normal DXF group code 0 cannot be used because DXF proxies use it to determine when to stop reading data. The group code 100 could have been used, but it might have caused confusion when manually reading a DXF file, and there was a need to distinguish when an embedded object is about to be written out in order to do some internal bookkeeping. Therefore, the DXF group code 101 was introduced.

Hard facts:

- Only used in ATTRIB, ATTDEF (embedded MTEXT) and MTEXT (columns) in DXF R2018.
- Embedded object start with (101, "Embedded Object") tag
- Embedded object is appended to the encapsulated object
- Embedded object tags can contain any group code except the DXF structure tag (0, ...)

Unconfirmed assumptions:

- The embedded object is written before the *Extended Data*. No examples for entities including embedded objects and XDATA at the same time.
- XDATA sections replaced by embedded objects, at least for the MTEXT entity
- The encapsulating object can contain more than one embedded object.
- Embedded objects separated by (101, "Embedded Object") tags
- every entity can contain embedded objects

Real world example from an AutoCAD 2018 file:

```
100      <<< start of encapsulating object
AcDbMText
10
2762.148
20
2327.073
30
0.0
40
2.5
41
18.852
46
0.0
71
1
72
5
1
{\fArial\b0|i0|c162|p34;CHANGE;\P\P\PTEXT}
73
1
44
1.0
101      <<< start of embedded object
Embedded Object
70
1
10
1.0
20
0.0
30
0.0
11
2762.148
21
2327.073
31
0.0
40
18.852
41
0.0
42
15.428
43
15.043
71
2
72
1
44
18.852
45
12.5
```

(continues on next page)

(continued from previous page)

73
0
74
0
46
0.0

Handles

A handle is an arbitrary but in your DXF file unique hex value as string like ‘10FF’. It is common to use uppercase letters for hex numbers. Handle can have up to 16 hexadecimal digits (8 bytes).

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

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

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

Handle Definition

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

Handle Pointer

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

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

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

Pointer and Ownership

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

Hard and Soft References

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

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

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

Arbitrary Handles

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

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

About 1005 Group Codes

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

DXF File Structure

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

See also:

For more information about DXF tags see: [DXF Tags](#)

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

1. **HEADER:** General information about the drawing is found in this section of the DXF file. Each parameter has a variable name starting with '\$' and an associated value. Has to be the first section.
2. **CLASSES:** Holds the information for application defined classes. (DXF R13 and later)
3. **TABLES:** Contains several tables for style and property definitions.
 - Linetype table (LTYPE)
 - Layer table (LAYER)
 - Text Style table (STYLE)
 - View table (VIEW): (IMHO) layout of the CAD working space, only interesting for interactive CAD applications
 - Viewport configuration table (VPORT): The VPORT table is unique in that it may contain several entries with the same name (indicating a multiple-viewport configuration). The entries corresponding to the active viewport configuration all have the name *ACTIVE. The first such entry describes the current viewport.
 - Dimension Style table (DIMSTYLE)

- User Coordinate System table (UCS) (IMHO) only interesting for interactive CAD applications
 - Application Identification table (APPID): Table of names for all applications registered with a drawing.
 - Block Record table (BLOCK_RECORD) (DXF R13 and Later)
4. **BLOCKS:** Contains all block definitions. The block name *Model_Space or *MODEL_SPACE is reserved for the drawing modelspace and the block name *Paper_Space or *PAPER_SPACE is reserved for the *active* paperspace layout. Both block definitions are empty, the content of the modelspace and the *active* paperspace is stored in the ENTITIES section. The entities of other layouts are stored in special block definitions called *Paper_Spacennn, nnn is an arbitrary but unique number.
 5. **ENTITIES:** Contains all graphical entities of the modelspace and the *active* paperspace layout. Entities of other layouts are stored in the BLOCKS sections.
 6. **OBJECTS:** Contains all non-graphical objects of the drawing (DXF R13 and later)
 7. **THUMBNAILIMAGE:** Contains a preview image of the DXF file, it is optional and can usually be ignored. (DXF R13 and later)
 8. **ACDS DATA:** (DXF R2013 and later) No information in the DXF reference about this section
 9. **END OF FILE**

For further information read the original [DXF Reference](#).

Structure of a usual DXF R12 file:

```

0           <<< Begin HEADER section, has to be the first section
SECTION
2
HEADER
        <<< Header variable items go here
0           <<< End HEADER section
ENDSEC
0           <<< Begin TABLES section
SECTION
2
TABLES
0
TABLE
2
VPORT
70           <<< viewport table maximum item count
                <<< viewport table items go here
0
ENDTAB
0
TABLE
2
APPID, DIMSTYLE, LTYPE, LAYER, STYLE, UCS, VIEW, or VPORT
70           <<< Table maximum item count, a not reliable value and ignored by AutoCAD
                <<< Table items go here
0
ENDTAB
0           <<< End TABLES section
ENDSEC
0           <<< Begin BLOCKS section
SECTION
2
BLOCKS

```

(continues on next page)

(continued from previous page)

```
        <<< Block definition entities go here
0      <<< End BLOCKS section
ENDSEC
0      <<< Begin ENTITIES section
SECTION
2
ENTITIES
        <<< Drawing entities go here
0      <<< End ENTITIES section
ENDSEC
0      <<< End of file marker (required)
EOF
```

Minimal DXF Content

DXF R12

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

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

DXF R13/R14 and later

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

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

The HEADER section requires two entries:

- \$ACADVER
- \$HANDSEED

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

The TABLES section requires following tables:

- VPORT entry *ACTIVE is not required! Empty table is ok for AutoCAD.
- LTYPE with at least the following line types defined:
 - BYBLOCK
 - BYLAYER
 - CONTINUOUS
- LAYER with at least an entry for layer ‘0’
- STYLE with at least an entry for style STANDARD
- VIEW can be empty

- UCS can be empty
- APPID with at least an entry for ACAD
- DIMSTYLE with at least an entry for style STANDARD
- BLOCK_RECORDS with two entries:
 - *MODEL_SPACE
 - *PAPER_SPACE

The BLOCKS section requires two BLOCKS:

- *MODEL_SPACE
- *PAPER_SPACE

The ENTITIES section can be empty.

The OBJECTS section requires following entities:

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

Minimal DXF to download: https://github.com/mozman/ezdxf/tree/master/examples_dxf

Data Model

Database Objects

(from the DXF Reference)

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

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

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

DXF R12 Data Model

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

- HEADER section: common settings for the DXF drawing
- TABLES section: definitions for LAYERS, LINETYPE, STYLES
- BLOCKS section: block definitions and its content
- ENTITIES section: modelspace and paperspace content

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

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

TODO: list of available entities

See also:

More information about the DXF [DXF File Structure](#)

DXF R13+ Data Model

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

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

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

The TABLES section got a new BLOCK_RECORD table - see [Block Management Structures](#) for more information.

The BLOCKS sections is mostly the same, but with handles, owner tags and new ENTITY types. Not active paperspace layouts store their content also in the BLOCKS section - see [Layout Management Structures](#) for more information.

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

TODO: list of new available entities

And the new OBJECTS section - now its getting complicated!

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

The OBJECTS sections stores all the non-graphical entities of the DXF drawing. Non-graphical entities from now on just called ‘DXF objects’ to differentiate them from graphical entities, just called ‘entities’. The OBJECTS section follows commonly the ENTITIES section, but this is not mandatory.

DXF R13 introduces several new DXF objects, which resides exclusive in the OBJECTS section, taken from the DXF R14 reference, because I have no access to the DXF R13 reference, the DXF R13 reference is a compiled .hlp file which can’t be read on Windows 10, a drastic real world example why it is better to avoid closed (proprietary) data formats ;):

- DICTIONARY: a general structural entity as a <name: handle> container
- ACDBDICTIONARYWDFLT: a DICTIONARY with a default value
- DICTIONARYVAR: used by AutoCAD to store named values in the database
- ACAD_PROXY_OBJECT: proxy object for entities created by other applications than AutoCAD
- GROUP: groups graphical entities without the need of a BLOCK definition
- IDBUFFER: just a list of references to objects

- IMAGEDEF: IMAGE definition structure, required by the IMAGE entity
- IMAGEDEF_REACTOR: also required by the IMAGE entity
- LAYER_INDEX: container for LAYER names
- MLINESTYLE
- OBJECT_PTR
- RASTERVARIABLES
- SPATIAL_INDEX: is always written out empty to a DXF file. This object can be ignored.
- SPATIAL_FILTER
- SORTESTTABLE: control for regeneration/redraw order of entities
- XRECORD: used to store and manage arbitrary data. This object is similar in concept to XDATA but is not limited by size or order. Not supported by R13c0 through R13c3.

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

New objects introduced by DXF R2000:

- LAYOUT: management object for modelspace and multiple paperspace layouts
- ACDBPLACEHOLDER: surprise - just a place holder

New objects in DXF R2004:

- DIMASSOC
- LAYER_FILTER
- MATERIAL
- PLOTSETTINGS
- VBA_PROJECT

New objects in DXF R2007:

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

- WIPEOUTVARIABLES

New objects in DXF R2013:

- GEODATA

New objects in DXF R2018:

- ACDBNAVISWORKSMODELDEF

Undocumented objects:

- SCALE
- ACDBSECTIONVIEWSTYLE
- FIELDLIST

Objects Organisation

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

- ACAD_COLOR: points to an empty DICTIONARY
- ACAD_GROUP: supported by ezdxf
- ACAD_LAYOUT: supported by ezdxf
- ACAD_MATERIAL: points to an empty DICTIONARY
- ACAD_MLEADERSTYLE: points to an empty DICTIONARY
- ACAD_MLINESTYLE: points to an empty DICTIONARY
- ACAD_PLOTSETTINGS: points to an empty DICTIONARY
- ACAD_PLOTSTYLENAME: points to ACDBDICTIONARYWDFLT with one entry: ‘Normal’
- ACAD_SCALELIST: points to an empty DICTIONARY
- ACAD_TABLESTYLE: points to an empty DICTIONARY
- ACAD_VISUALSTYLE: points to an empty DICTIONARY

Root DICTIONARY content for DXF R2018

```
0
SECTION
2      <<< start of the OBJECTS section
OBJECTS
0      <<< root DICTIONARY has to be the first object in the OBJECTS section
DICTIONARY
5      <<< handle
C
330    <<< owner tag
0      <<< always #0, has no owner
```

(continues on next page)

(continued from previous page)

```

100
AcDbDictionary
281      <<< hard owner flag
1
3      <<< first entry
ACAD_CIP_PREVIOUS_PRODUCT_INFO
350      <<< handle to target (pointer)
78B      <<< points to a XRECORD with product info about the creator application
3      <<< entry with unknown meaning, if I shoul guess: something with about colors_
↪...
ACAD_COLOR
350
4FB      <<< points to a DICTIONARY
3      <<< entry with unknown meaning
ACAD_DETAILVIEWSTYLE
350
7ED      <<< points to a DICTIONARY
3      <<< GROUP management, mandatory in all DXF versions
ACAD_GROUP
350
4FC      <<< points to a DICTIONARY
3      <<< LAYOUT management, mandatory if more than the *active* paperspace is used
ACAD_LAYOUT
350
4FD      <<< points to a DICTIONARY
3      <<< MATERIAL management
ACAD_MATERIAL
350
4FE      <<< points to a DICTIONARY
3      <<< MLEADERSTYLE management
ACAD_MLEADERSTYLE
350
4FF      <<< points to a DICTIONARY
3      <<< MLINESTYLE management
ACAD_MLINESTYLE
350
500      <<< points to a DICTIONARY
3      <<< PLOTSETTINGS management
ACAD_PLOTSETTINGS
350
501      <<< points to a DICTIONARY
3      <<< plot style name management
ACAD_PLOTSTYLENAME
350
503      <<< points to a ACDBDICTIONARYWDFLT
3      <<< SCALE management
ACAD_SCALELIST
350
504      <<< points to a DICTIONARY
3      <<< entry with unknown meaning
ACAD_SECTIONVIEWSTYLE
350
7EB      <<< points to a DICTIONARY
3      <<< TABLESTYLE management
ACAD_TABLESTYLE
350
505      <<< points to a DICTIONARY

```

(continues on next page)

(continued from previous page)

```
3      <<< VISUALSTYLE management
ACAD_VISUALSTYLE
350
506      <<< points to a DICTIONARY
3      <<< entry with unknown meaning
ACDB_RECOMPOSE_DATA
350
7F3
3      <<< entry with unknown meaning
AcDbVariableDictionary
350
7AE      <<< points to a DICTIONARY with handles to DICTIONARYVAR objects
0
DICTIONARY
...
...
0
ENDSEC
```

6.12.2 DXF Structures

DXF Sections

HEADER Section

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

```
0      <<< Begin HEADER section
SECTION
2
HEADER
9
$ACADVER    <<< Header variable items go here
1
AC1009
...
0
ENDSEC    <<< End HEADER section
```

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

See also:

Documentation of *ezdxf HeaderSection* class.

DXF Reference: [Header Variables](#)

CLASSES Section

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

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

See also:

DXF Reference: [About the DXF CLASSES Section](#)

Documentation of `ezdxf.ClassesSection` class.

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

Update 2019-03-03:

Class names are not unique, Autodesk Architectural Desktop 2007 uses the same name, but with different CPP class names in the CLASS section, so storing classes in a dictionary by name as key caused loss of class entries in ezdxf, using a tuple of (name, cpp_class_name) as storage key solved the problem.

CLASS Entities**See also:**

DXF Reference: [Group Codes for the CLASS entity](#)

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

```

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

```

TABLES Section

TODO

BLOCKS Section

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

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

BLOCKS section structure:

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

See also:

[Block Management Structures](#) [Layout Management Structures](#)

ENTITIES Section

TODO

OBJECTS Section

Objects in the OBJECTS section are organized in a hierarchical tree order, starting with the *named objects dictionary* as the first entity in the OBJECTS section (`Drawing.rootdict`).

Not all entities in the OBJECTS section are included in this tree, [Extension Dictionary](#) and XRECORD data of graphical entities are also stored in the OBJECTS section.

DXF Tables

VIEW Table

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

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

DXF R12

```

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

```

See also:

Coordinate Systems

DXF R2000+

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

```
0      <<< adding the VIEW table head, just for information
TABLE
2      <<< table name
VIEW
5      <<< handle of table, see owner tag of VIEW table entry
37C
330    <<< owner tag of table, always #0
0
100    <<< subclass marker
AcDbSymbolTable
70     <<< VIEW table (max.) count, not reliable (ignore)
9
0      <<< first VIEW table entry
VIEW
5      <<< handle
3EA
330    <<< owner, the VIEW table is the owner of the VIEW entry
37C    <<< handle of the VIEW table
100    <<< subclass marker
AcDbSymbolTableRecord
100    <<< subclass marker
AcDbViewTableRecord
2      <<< view name, from here all the same as DXF R12
VIEWNAME
70
0
40
20.01
10
40.36
20
15.86
41
17.91
11
0.0
21
0.0
31
1.0
12
0.0
22
0.0
32
0.0
42
50.0
43
0.0
44
0.0
50
```

(continues on next page)

(continued from previous page)

```

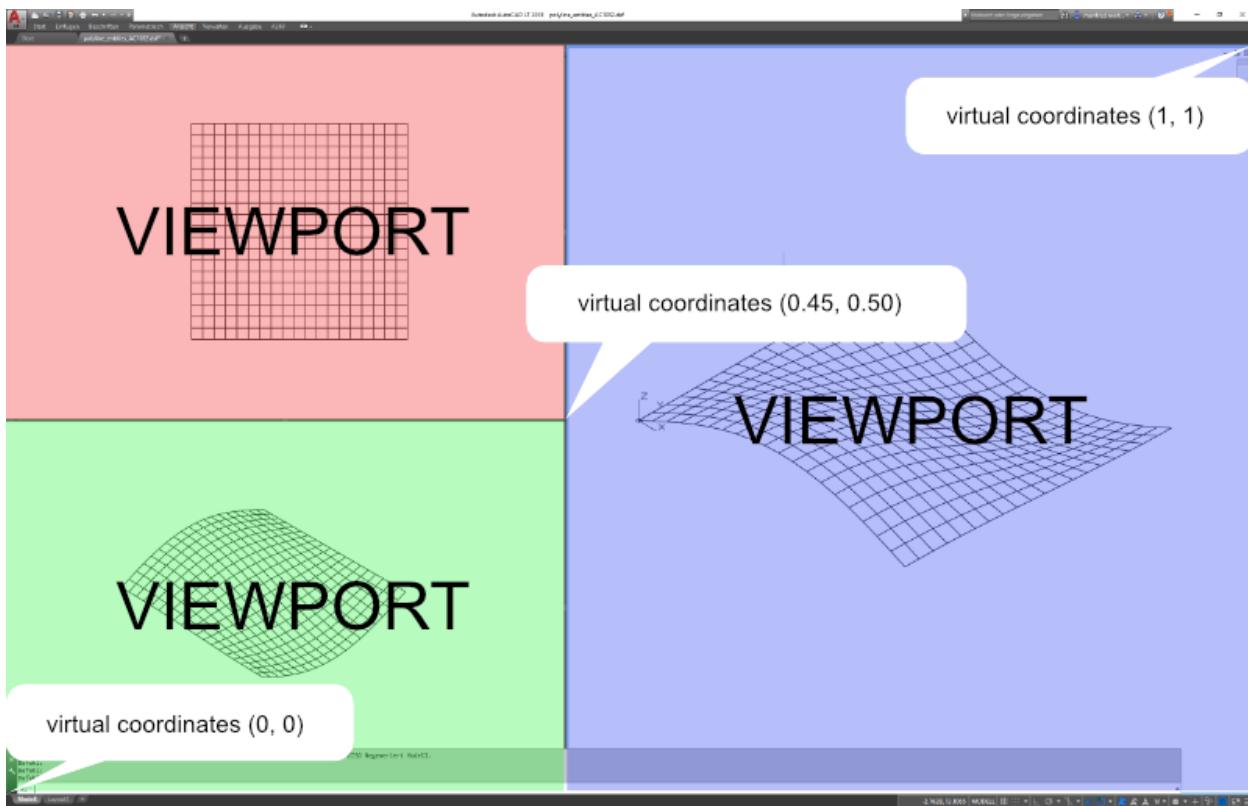
0.0
71
0
281    <<< render mode 0-6 (... too much options)
0      <<< 0= 2D optimized (classic 2D)
72      <<< UCS associated (0/1 = no/yes)
0      <<< 0 = no

```

DXF R2000+ supports additional features in the VIEW entry, see the [VIEW](#) table reference provided by Autodesk.

VPORT Configuration Table

The [VPORT](#) table stores the modelspace viewport configurations. A viewport configuration is a tiled view of multiple viewports or just one viewport.



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

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

The following statements refer to a 2D plan view: the view-target-point defines the origin of the DCS (Display Coordinate system), the view-direction vector defines the z-axis of the [DCS](#), the view-center-point (in DCS) defines the point in modelspace translated to the center point of the viewport, the view height and the aspect-ratio defines how much of the modelspace is displayed. AutoCAD tries to fit the modelspace area into the available viewport space e.g.

view height is 15 units and aspect-ratio is 2.0 the modelspace to display is 30 units wide and 15 units high, if the viewport has an aspect ratio of 1.0, AutoCAD displays 30x30 units of the modelspace in the viewport. If the modelspace aspect-ratio is 1.0 the modelspace to display is 15x15 units and fits properly into the viewport area.

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

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

DXF R12

Multi-viewport configuration with three viewports.

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

(continues on next page)

(continued from previous page)

```

17      <<< view target point (in WCS), defines the origin of the DCS
0.0      <<<     x value
27      <<<     group code for y value
0.0      <<<     y value
37      <<<     group code for z value
0.0      <<<     z value
40      <<< view height
35.22
41      <<< viewport aspect ratio
0.99
42      <<< lens (focal) length
50.0      <<< 50mm
43      <<< front clipping planes, offsets from target point
0.0
44      <<< back clipping planes, offsets from target point
0.0
50      <<< snap rotation angle
0.0
51      <<< view twist angle
0.0
71      <<< view mode
0
72      <<< circle zoom percent
1000
73      <<< fast zoom setting
1
74      <<< UCSICON setting
3
75      <<< snap on/off
0
76      <<< grid on/off
0
77      <<< snap style
0
78      <<< snap isopair
0
0      <<< next VPORT entry
VPORT
2      <<< VPORT (configuration) name
*ACTIVE <<< same as first VPORT entry
70
0
10
0.0
20
0.5
11
0.45
21
1.0
12
8.21
22
9.41
...
...
0      <<< next VPORT entry

```

(continues on next page)

(continued from previous page)

```

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

```

DXF R2000+

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

```

0      <<< table start
TABLE
2      <<< table type
VPORT
5      <<< table handle
151F
330    <<< owner, table has no owner - always #0
0
100    <<< subclass marker
AcDbSymbolTable
70     <<< VPORT table (max.) count, not reliable (ignore)
3
0      <<< first VPORT entry
VPORT
5      <<< entry handle
158B
330    <<< owner, VPORT table is owner of VPORT entry
151F
100    <<< subclass marker
AcDbSymbolTableRecord
100    <<< subclass marker
AcDbViewportTableRecord
2      <<< VPORT (configuration) name
*ACTIVE
70     <<< standard flags, bit-coded
0
10    <<< lower-left corner of viewport
0.45   <<<     x value, virtual coordinates in range [0 - 1]
20     <<<     group code for y value

```

(continues on next page)

(continued from previous page)

```

0.0      <<<      y value, virtual coordinates in range [0 - 1]
11       <<<      upper-right corner of viewport
1.0      <<<      x value, virtual coordinates in range [0 - 1]
21       <<<      group code for y value
1.0      <<<      y value, virtual coordinates in range [0 - 1]
12       <<<      view center point (in DCS)
13.71    <<<      x value
22       <<<      group code for y value
0.38    <<<      y value
13       <<<      snap base point (in DCS)
0.0      <<<      x value
23       <<<      group code for y value
0.0      <<<      y value
14       <<<      snap spacing X and Y
1.0      <<<      x value
24       <<<      group code for y value
1.0      <<<      y value
15       <<<      grid spacing X and Y
0.0      <<<      x value
25       <<<      group code for y value
0.0      <<<      y value
16       <<<      view direction from target point (in WCS)
1.0      <<<      x value
26       <<<      group code for y value
-1.0     <<<      y value
36       <<<      group code for z value
1.0      <<<      z value
17       <<<      view target point (in WCS)
0.0      <<<      x value
27       <<<      group code for y value
0.0      <<<      y value
37       <<<      group code for z value
0.0      <<<      z value
40       <<<      view height
35.22   <<<
41       <<<      viewport aspect ratio
0.99
42       <<<      lens (focal) length
50.0    <<<      50mm
43       <<<      front clipping planes, offsets from target point
0.0
44       <<<      back clipping planes, offsets from target point
0.0
50       <<<      snap rotation angle
0.0
51       <<<      view twist angle
0.0
71       <<<      view mode
0
72       <<<      circle zoom percent
1000
73       <<<      fast zoom setting
1
74       <<<      UCSICON setting
3
75       <<<      snap on/off
0

```

(continues on next page)

(continued from previous page)

```

76      <<< grid on/off
0
77      <<< snap style
0
78      <<< snap isopair
0
281      <<< render mode 1-6 (... too many options)
0          <<< 0 = 2D optimized (classic 2D)
65          <<< Value of UCSVP for this viewport. (0 = UCS will not change when this
      ↵viewport is activated)
1          <<< 1 = then viewport stores its own UCS which will become the current UCS
      ↵whenever the viewport is activated.
110      <<< UCS origin (3D point)
0.0      <<<     x value
120      <<<     group code for y value
0.0      <<<     y value
130      <<<     group code for z value
0.0      <<<     z value
111      <<< UCS X-axis (3D vector)
1.0      <<<     x value
121      <<<     group code for y value
0.0      <<<     y value
131      <<<     group code for z value
0.0      <<<     z value
112      <<< UCS Y-axis (3D vector)
0.0      <<<     x value
122      <<<     group code for y value
1.0      <<<     y value
132      <<<     group code for z value
0.0      <<<     z value
79      <<< Orthographic type of UCS 0-6 (... too many options)
0          <<< 0 = UCS is not orthographic
146          <<< elevation
0.0
1001     <<< extended data - undocumented
ACAD_NAV_VCDISPLAY
1070
3
0      <<< next VPORT entry
VPORT
5
158C
330
151F
100
AcDbSymbolTableRecord
100
AcDbViewportTableRecord
2          <<< VPORT (configuration) name
*ACTIVE <<< same as first VPORT entry
70
0
10
0.0
20
0.5
11

```

(continues on next page)

(continued from previous page)

```

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

```

LTYPE Table

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

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

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

See also:

- DXF Reference: [TABLES Section](#)
- DXF Reference: [LTYPE Table](#)

Table Structure DXF R12

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

Table Structure DXF R2000+

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

Simple Line Type

ezdxf setup for line type ‘CENTER’:

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

Simple Line Type Tag Structure DXF R2000+

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

Complex Line Type TEXT

ezdxf setup for line type ‘GASLEITUNG’:

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

TEXT Tag Structure

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

Complex Line Type SHAPE

ezdxf setup for line type ‘GRENZE2’:

```

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

```

SHAPE Tag Structure

```

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

```

(continues on next page)

(continued from previous page)

1 . 0
74
0

DIMSTYLE Table

The **DIMSTYLE** table stores all dimension style definitions of a DXF drawing.

You have access to the dimension styles table by the attribute `Drawing.dimstyles`.

See also:

- DXF Reference: [TABLES Section](#)
- DXF Reference: [DIMSTYLE Table](#)

Table Structure DXF R12

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

DIMSTYLE Entry DXF R12

DIMSTYLE Variables DXF R12

Source: [CADDManager Blog](#)

DIMVAR	Code	Description
DIMALT	170	Controls the display of alternate units in dimensions.

Continued on next page

Table 4 – continued from previous page

DIMVAR	Code	Description
DIMALTD	171	Controls the number of decimal places in alternate units. If DIMALT is turned on, DIMALTD sets the number of digits displayed to the right of the decimal point in the alternate measurement.
DIMALTF	143	Controls the multiplier for alternate units. If DIMALT is turned on, DIMALTF multiplies linear dimensions by a factor to produce a value in an alternate system of measurement. The initial value represents the number of millimeters in an inch.
DIMAPOST	4	Specifies a text prefix or suffix (or both) to the alternate dimension measurement for all types of dimensions except angular. For instance, if the current units are Architectural, DIMALT is on, DIMALTF is 25.4 (the number of millimeters per inch), DIMALTD is 2, and DIMPOST is set to “mm”, a distance of 10 units would be displayed as 10”[254.00mm].
DIMASZ	41	Controls the size of dimension line and leader line arrowheads. Also controls the size of hook lines. Multiples of the arrowhead size determine whether dimension lines and text should fit between the extension lines. DIMASZ is also used to scale arrowhead blocks if set by DIMBLK. DIMASZ has no effect when DIMTSZ is other than zero.
DIMBLK	5	Sets the arrowhead block displayed at the ends of dimension lines.
DIMBLK1	6	Sets the arrowhead for the first end of the dimension line when DIMSAH is 1.
DIMBLK2	7	Sets the arrowhead for the second end of the dimension line when DIMSAH is 1.

Continued on next page

Table 4 – continued from previous page

DIMVAR	Code	Description
DIMCEN	141	Controls drawing of circle or arc center marks and centerlines by the DIMCENTER, DIMDIAMETER, and DIMRADIUS commands. For DIMDIAMETER and DIMRADIUS, the center mark is drawn only if you place the dimension line outside the circle or arc. <ul style="list-style-type: none"> • 0 = No center marks or lines are drawn • <0 = Centerlines are drawn • >0 = Center marks are drawn
DIMCLRD	176	Assigns colors to dimension lines, arrowheads, and dimension leader lines. <ul style="list-style-type: none"> • 0 = BYBLOCK • 1-255 = ACI AutoCAD Color Index • 256 = BYLAYER
DIMCLRE	177	Assigns colors to dimension extension lines, values like DIMCLRD
DIMCLRT	178	Assigns colors to dimension text, values like DIMCLRD
DIMDLE	46	Sets the distance the dimension line extends beyond the extension line when oblique strokes are drawn instead of arrowheads.
DIMDLI	43	Controls the spacing of the dimension lines in baseline dimensions. Each dimension line is offset from the previous one by this amount, if necessary, to avoid drawing over it. Changes made with DIMDLI are not applied to existing dimensions.
DIMEXE	44	Specifies how far to extend the extension line beyond the dimension line.
DIMEXO	42	Specifies how far extension lines are offset from origin points. With fixed-length extension lines, this value determines the minimum offset.

Continued on next page

Table 4 – continued from previous page

DIMVAR	Code	Description
DIMGAP	147	<p>Sets the distance around the dimension text when the dimension line breaks to accommodate dimension text. Also sets the gap between annotation and a hook line created with the LEADER command. If you enter a negative value, DIMGAP places a box around the dimension text.</p> <p>DIMGAP is also used as the minimum length for pieces of the dimension line. When the default position for the dimension text is calculated, text is positioned inside the extension lines only if doing so breaks the dimension lines into two segments at least as long as DIMGAP. Text placed above or below the dimension line is moved inside only if there is room for the arrowheads, dimension text, and a margin between them at least as large as DIMGAP: $2 * (\text{DIMASZ} + \text{DIMGAP})$.</p>
DIMLFAC	144	<p>Sets a scale factor for linear dimension measurements. All linear dimension distances, including radii, diameters, and coordinates, are multiplied by DIMLFAC before being converted to dimension text. Positive values of DIMLFAC are applied to dimensions in both modelspace and paperspace; negative values are applied to paperspace only.</p> <p>DIMLFAC applies primarily to nonassociative dimensions (DIMASSOC set 0 or 1). For nonassociative dimensions in paperspace, DIMLFAC must be set individually for each layout viewport to accommodate viewport scaling.</p> <p>DIMLFAC has no effect on angular dimensions, and is not applied to the values held in DIMRND, DIMTM, or DIMTP.</p>

Continued on next page

Table 4 – continued from previous page

DIMVAR	Code	Description
DIMLIM	72	<p>Generates dimension limits as the default text. Setting DIMLIM to On turns DIMTOL off.</p> <ul style="list-style-type: none"> • 0 = Dimension limits are not generated as default text • 1 = Dimension limits are generated as default text
DIMPOST	3	<p>Specifies a text prefix or suffix (or both) to the dimension measurement.</p> <p>For example, to establish a suffix for millimeters, set DIMPOST to mm; a distance of 19.2 units would be displayed as 19.2 mm. If tolerances are turned on, the suffix is applied to the tolerances as well as to the main dimension.</p> <p>Use “<>” to indicate placement of the text in relation to the dimension value. For example, enter “<>mm” to display a 5.0 millimeter radial dimension as “5.0mm”. If you entered mm “<>”, the dimension would be displayed as “mm 5.0”.</p>
DIMRND	45	<p>Rounds all dimensioning distances to the specified value.</p> <p>For instance, if DIMRND is set to 0.25, all distances round to the nearest 0.25 unit. If you set DIMRND to 1.0, all distances round to the nearest integer. Note that the number of digits edited after the decimal point depends on the precision set by DIMDEC. DIMRND does not apply to angular dimensions.</p>
DIMSAH	173	<p>Controls the display of dimension line arrowhead blocks.</p> <ul style="list-style-type: none"> • 0 = Use arrowhead blocks set by DIMBLK • 1 = Use arrowhead blocks set by DIMBLK1 and DIMBLK2

Continued on next page

Table 4 – continued from previous page

DIMVAR	Code	Description
DIMSCALE	40	<p>Sets the overall scale factor applied to dimensioning variables that specify sizes, distances, or offsets. Also affects the leader objects with the LEADER command.</p> <p>Use MLEADERSCALE to scale multileader objects created with the MLEADER command.</p> <ul style="list-style-type: none"> • 0.0 = A reasonable default value is computed based on the scaling between the current model space viewport and paperspace. If you are in paperspace or modelspace and not using the paperspace feature, the scale factor is 1.0. • >0 = A scale factor is computed that leads text sizes, arrowhead sizes, and other scaled distances to plot at their face values. <p>DIMSCALE does not affect measured lengths, coordinates, or angles.</p> <p>Use DIMSCALE to control the overall scale of dimensions. However, if the current dimension style is annotative, DIMSCALE is automatically set to zero and the dimension scale is controlled by the CANNOSCALE system variable. DIMSCALE cannot be set to a non-zero value when using annotative dimensions.</p>
DIMSE1	75	<p>Suppresses display of the first extension line.</p> <ul style="list-style-type: none"> • 0 = Extension line is not suppressed • 1 = Extension line is suppressed
DIMSE2	76	<p>Suppresses display of the second extension line.</p> <ul style="list-style-type: none"> • 0 = Extension line is not suppressed • 1 = Extension line is suppressed

Continued on next page

Table 4 – continued from previous page

DIMVAR	Code	Description
DIMSOXD	175	<p>Suppresses arrowheads if not enough space is available inside the extension lines.</p> <ul style="list-style-type: none"> • 0 = Arrowheads are not suppressed • 1 = Arrowheads are suppressed <p>If not enough space is available inside the extension lines and DIMTIX is on, setting DIMSOXD to On suppresses the arrowheads. If DIMTIX is off, DIMSOXD has no effect.</p>
DIMTAD	77	<p>Controls the vertical position of text in relation to the dimension line.</p> <ul style="list-style-type: none"> • 0 = Centers the dimension text between the extension lines. • 1 = Places the dimension text above the dimension line except when the dimension line is not horizontal and text inside the extension lines is forced horizontal (DIMTIH = 1). The distance from the dimension line to the baseline of the lowest line of text is the current DIMGAP value. • 2 = Places the dimension text on the side of the dimension line farthest away from the defining points. • 3 = Places the dimension text to conform to Japanese Industrial Standards (JIS). • 4 = Places the dimension text below the dimension line.
DIMTFAC	146	<p>Specifies a scale factor for the text height of fractions and tolerance values relative to the dimension text height, as set by DIMTXT.</p> <p>For example, if DIMTFAC is set to 1.0, the text height of fractions and tolerances is the same height as the dimension text. If DIMTFAC is set to 0.7500, the text height of fractions and tolerances is three-quarters the size of dimension text.</p>

Continued on next page

Table 4 – continued from previous page

DIMVAR	Code	Description
DIMTIH	73	Controls the position of dimension text inside the extension lines for all dimension types except Ordinate. <ul style="list-style-type: none"> • 0 = Aligns text with the dimension line • 1 = Draws text horizontally
DIMTIX	174	Draws text between extension lines. <ul style="list-style-type: none"> • 0 = Varies with the type of dimension. For linear and angular dimensions, text is placed inside the extension lines if there is sufficient room. For radius and diameter dimensions hat don't fit inside the circle or arc, DIMTIX has no effect and always forces the text outside the circle or arc. • 1 = Draws dimension text between the extension lines even if it would ordinarily be placed outside those lines
DIMTM	48	Sets the minimum (or lower) tolerance limit for dimension text when DIMTOL or DIMLIM is on. DIMTM accepts signed values. If DIMTOL is on and DIMTP and DIMTM are set to the same value, a tolerance value is drawn. If DIMTM and DIMTP values differ, the upper tolerance is drawn above the lower, and a plus sign is added to the DIMTP value if it is positive. For DIMTM, the program uses the negative of the value you enter (adding a minus sign if you specify a positive number and a plus sign if you specify a negative number).

Continued on next page

Table 4 – continued from previous page

DIMVAR	Code	Description
DIMTOFL	172	<p>Controls whether a dimension line is drawn between the extension lines even when the text is placed outside. For radius and diameter dimensions (when DIMTIX is off), draws a dimension line inside the circle or arc and places the text, arrowheads, and leader outside.</p> <ul style="list-style-type: none"> • 0 = Does not draw dimension lines between the measured points when arrowheads are placed outside the measured points • 1 = Draws dimension lines between the measured points even when arrowheads are placed outside the measured points
DIMTOH	74	<p>Controls the position of dimension text outside the extension lines.</p> <ul style="list-style-type: none"> • 0 = Aligns text with the dimension line • 1 = Draws text horizontally
DIMTOL	71	Appends tolerances to dimension text. Setting DIMTOL to on turns DIMLIM off.
DIMTP	47	Sets the maximum (or upper) tolerance limit for dimension text when DIMTOL or DIMLIM is on. DIMTP accepts signed values. If DIMTOL is on and DIMTP and DIMTM are set to the same value, a tolerance value is drawn. If DIMTM and DIMTP values differ, the upper tolerance is drawn above the lower and a plus sign is added to the DIMTP value if it is positive.
DIMTSZ	142	<p>Specifies the size of oblique strokes drawn instead of arrowheads for linear, radius, and diameter dimensioning.</p> <ul style="list-style-type: none"> • 0 = Draws arrowheads. • >0 = Draws oblique strokes instead of arrowheads. The size of the oblique strokes is determined by this value multiplied by the DIMSCALE value

Continued on next page

Table 4 – continued from previous page

DIMVAR	Code	Description
DIMTVP	145	Controls the vertical position of dimension text above or below the dimension line. The DIMTVP value is used when DIMTAD = 0. The magnitude of the vertical offset of text is the product of the text height and DIMTVP. Setting DIMTVP to 1.0 is equivalent to setting DIMTAD = 1. The dimension line splits to accommodate the text only if the absolute value of DIMTVP is less than 0.7.
DIMTXT	140	Specifies the height of dimension text, unless the current text style has a fixed height.
DIMZIN	78	Controls the suppression of zeros in the primary unit value. Values 0-3 affect feet-and-inch dimensions only: <ul style="list-style-type: none"> • 0 = Suppresses zero feet and precisely zero inches • 1 = Includes zero feet and precisely zero inches • 2 = Includes zero feet and suppresses zero inches • 3 = Includes zero inches and suppresses zero feet • 4 (Bit 3) = Suppresses leading zeros in decimal dimensions (for example, 0.5000 becomes .5000) • 8 (Bit 4) = Suppresses trailing zeros in decimal dimensions (for example, 12.5000 becomes 12.5) • 12 (Bit 3+4) = Suppresses both leading and trailing zeros (for example, 0.5000 becomes .5)

Table Structure DXF R2000+

```

0      <<< start of table
TABLE
2      <<< set table type
DIMSTYLE
5      <<< DIMSTYLE table handle
5F
330    <<< owner tag, tables has no owner
0

```

(continues on next page)

(continued from previous page)

```
100      <<< subclass marker
AcDbSymbolTable
70       <<< count of dimension styles defined in this table, AutoCAD ignores this _
_value
9
0       <<< 1. DIMSTYLE table entry
DIMSTYLE
        <<< DIMSTYLE data tags
0       <<< 2. DIMSTYLE table entry
DIMSTYLE
        <<< DIMSTYLE data tags and so on
0       <<< end of DIMSTYLE table
ENDTAB
```

Additional DIMSTYLE Variables DXF R13/14

Source: [CADDManager Blog](#)

DIMVAR	code	Description
DIMADEC	179	Controls the number of precision places displayed in angular dimensions.
DIMALTTD	274	Sets the number of decimal places for the tolerance values in the alternate units of a dimension.
DIMALTZ	286	Controls suppression of zeros in tolerance values.
DIMALTU	273	Sets the units format for alternate units of all dimension substyles except Angular.
DIMALTZ	285	Controls the suppression of zeros for alternate unit dimension values. DIMALTZ values 0-3 affect feet-and-inch dimensions only.
DIMAUNIT	275	Sets the units format for angular dimensions. <ul style="list-style-type: none"> • 0 = Decimal degrees • 1 = Degrees/minutes/seconds • 2 = Grad • 3 = Radians
DIMBLK_HANDLE	342	defines DIMBLK as handle to the BLOCK RECORD entry
DIMBLK1_HANDLE	343	defines DIMBLK1 as handle to the BLOCK RECORD entry
DIMBLK2_HANDLE	344	defines DIMBLK2 as handle to the BLOCK RECORD entry
DIMDEC	271	Sets the number of decimal places displayed for the primary units of a dimension. The precision is based on the units or angle format you have selected.
DIMDSEP	278	Specifies a single-character decimal separator to use when creating dimensions whose unit format is decimal. When prompted, enter a single character at the Command prompt. If dimension units is set to Decimal, the DIMDSEP character is used instead of the default decimal point. If DIMDSEP is set to NULL (default value, reset by entering a period), the decimal point is used as the dimension separator.
DIMJUST	280	Controls the horizontal positioning of dimension text. <ul style="list-style-type: none"> • 0 = Positions the text above the dimension line and center-justifies it between the extension lines • 1 = Positions the text next to the first extension line • 2 = Positions the text next to the second extension line 635 • 3 = Positions the text above and aligned with the first extension line
6.12. DXF Internals		

Additional DIMSTYLE Variables DXF R2000

Source: [CADDManager Blog](#)

DIMVAR	Code	Description
DIMALTRND	148	Rounds off the alternate dimension units.
DIMATFIT	289	<p>Determines how dimension text and arrows are arranged when space is not sufficient to place both within the extension lines.</p> <ul style="list-style-type: none"> • 0 = Places both text and arrows outside extension lines • 1 = Moves arrows first, then text • 2 = Moves text first, then arrows • 3 = Moves either text or arrows, whichever fits best <p>A leader is added to moved dimension text when DIMTMOVE is set to 1.</p>
DIMAZIN	79	<p>Suppresses zeros for angular dimensions.</p> <ul style="list-style-type: none"> • 0 = Displays all leading and trailing zeros • 1 = Suppresses leading zeros in decimal dimensions (for example, 0.5000 becomes .5000) • 2 = Suppresses trailing zeros in decimal dimensions (for example, 12.5000 becomes 12.5) • 3 = Suppresses leading and trailing zeros (for example, 0.5000 becomes .5)
DIMFRAC	276	<p>Sets the fraction format when DIMLUNIT is set to 4 (Architectural) or 5 (Fractional).</p> <ul style="list-style-type: none"> • 0 = Horizontal stacking • 1 = Diagonal stacking • 2 = Not stacked (for example, 1/2)
DIMLDRBLK_HANDLE	341	Specifies the arrow type for leaders. Handle to BLOCK RECORD
DIMLUNIT	277	<p>Sets units for all dimension types except Angular.</p> <ul style="list-style-type: none"> • 1 = Scientific • 2 = Decimal • 3 = Engineering • 4 = Architectural (always displayed stacked) • 5 = Fractional (always displayed stacked) • 6 = Microsoft Windows Desktop (decimal format using Control Panel settings for decimal separator and number grouping symbols)
6.12. DXF Internals		

Text Location

This image shows the default text locations created by BricsCAD for dimension variables `dimtad` and `dimjust`:

Unofficial DIMSTYLE Variables for DXF R2007 and later

The following DIMVARS are **not documented** in the [DXF Reference](#) by Autodesk.

DIMVAR	Code	Description
DIMTFILL	69	Text fill 0=off; 1=background color; 2=custom color (see <code>DIMTFILLCLR</code>)
DIMTFILL-CLR	70	Text fill custom color as color index
DIMFXLON	290	Extension line has fixed length if set to 1
DIMFXL	49	Length of extension line below dimension line if fixed (<code>DIMFXLON</code> is 1), <code>DIMEXE</code> defines the the length above the dimension line
DIMJOGANG	50	Angle of oblique dimension line segment in jogged radius dimension
DIML-TYPE_HANDLE	345	Specifies the LINETYPE of the dimension line. Handle to LTYPE table entry
DIML-TEX1_HANDLE	346	Specifies the LINETYPE of the extension line 1. Handle to LTYPE table entry
DIML-TEX2_HANDLE	347	Specifies the LINETYPE of the extension line 2. Handle to LTYPE table entry

Extended Settings as Special XDATA Groups

Prior to DXF R2007, some extended settings for the dimension and the extension lines are stored in the XDATA section by following entries, this is not documented by Autodesk:

```
1001
ACAD_DSTYLE_DIM_LINETYPE      <<< linetype for dimension line
1070
380                           <<< group code, which differs from R2007 DIMDLTYPE
1005
FFFF                          <<< handle to LTYPE entry
1001
ACAD_DSTYLE_DIM_EXT1_LINETYPE <<< linetype for extension line 1
1070
381                           <<< group code, which differs from R2007 DIMLTEX1
1005
FFFF                          <<< handle to LTYPE entry
1001
ACAD_DSTYLE_DIM_EXT2_LINETYPE <<< linetype for extension line 1
1070
382                           <<< group code, which differs from R2007 DIMLTEX2
1005
FFFF                          <<< handle to LTYPE entry
1001
ACAD_DSTYLE_DIMEXT_ENABLED    <<< extension line fixed
1070
383                           <<< group code, which differs from R2007 DIMEXFIX
1070
```

(continues on next page)

(continued from previous page)

1	<<< fixed if 1 else 0
1001	
ACAD_DSTYLE_DIMEXT_LENGTH	<<< extension line fixed length
1070	
378	<<< group code, which differs from R2007 DIMEXLEN
1040	
1.33	<<< length of extension line below dimension line

This XDATA groups requires also an appropriate APPID entry in the APPID table. This feature is not supported by *ezdxf*.

BLOCK_RECORD Table

Block records are essential elements for the entities management, each layout (modelspace and paperspace) and every block definition has a block record entry. This block record is the hard *owner* of the entities of layouts, each entity has an owner handle which points to a block record of the layout.

DXF Entities

DIMENSION Internals

See also:

- DXF Reference: [DIMENSION](#)
- DXFInternals: [DIMSTYLE Table](#)

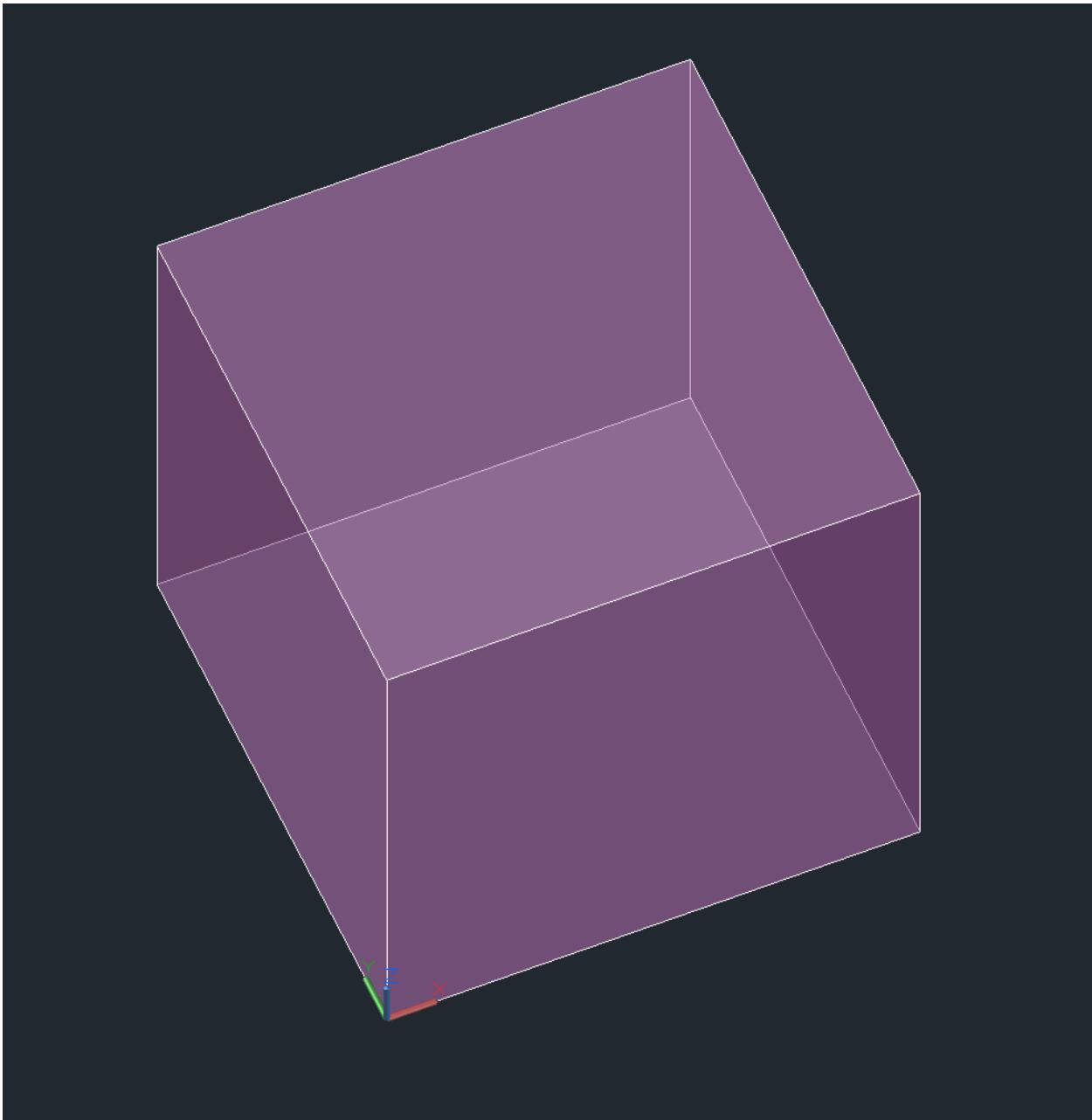
MESH Internals

The MESH entity is the compact version of the PolyFaceMesh implemented by the [Polyline](#) entity . The entity stores the vertices, edges and faces in a single entity and was introduced in DXF version R13/R14. For more information about the top level stuff go to the [Mesh](#) class.

See also:

- DXF Reference: [MESH](#)
- `ezdxf.entities.Mesh` class

The following DXF code represents this cube with subdivision level of 0:



```
0
MESH          <<< DXF type
5             <<< entity handle
2F
330           <<< block record handle of owner layout
17
100
AcDbEntity
8
0             <<< layer
62
6             <<< color
100
```

(continues on next page)

(continued from previous page)

```

AcDbSubDMesh    <<< subclass marker
71
2             <<< version
72
1             <<< blend crease, 1 is "on", 0 is "off"
91
0             <<< subdivision level is 0
92
8             <<< vertex count, 8 cube corners
10            <<< 1. vertex, x-axis
0.0
20            <<< y-axis
0.0
30            <<< z-axis
0.0
10            <<< 2. vertex
1.0
20
0.0
30
0.0
10            <<< 3. vertex
1.0
20
1.0
30
0.0
10            <<< 4. vertex
0.0
20
1.0
30
0.0
10            <<< 5. vertex
0.0
20
0.0
30
1.0
10            <<< 6. vertex
1.0
20
0.0
30
1.0
10            <<< 7. vertex
1.0
20
1.0
30
1.0
10            <<< 8. vertex
0.0
20
1.0
30
1.0

```

(continues on next page)

(continued from previous page)

```
93      <<< size of face list
30      <<< size = count of group code 90 tags = 6 x 5
90      <<< vertex count of face 1
4       <<< MESH supports ngons, count = 3, 4, 5, 6 ...
90
0       <<< face 1, index of 1. vertex
90
3       <<< face 1, index of 2. vertex
90
2       <<< face 1, index of 3. vertex
90
1       <<< face 1, index of 4. vertex
90
4           <<< vertex count of face 2
90
4       <<< face 2, index of 1. vertex
90
5       <<< face 2, index of 2. vertex
90
6       <<< face 2, index of 3. vertex
90
7       <<< face 2, index of 4. vertex
90
4           <<< vertex count of face 3
90
0       <<< face 3, index of 1. vertex
90
1       <<< face 3, index of 2. vertex
90
5       <<< face 3, index of 3. vertex
90
4       <<< face 3, index of 4. vertex
90
4           <<< vertex count of face 4
90
1       <<< face 4, index of 1. vertex
90
2       <<< face 4, index of 2. vertex
90
6       <<< face 4, index of 3. vertex
90
5       <<< face 4, index of 4. vertex
90
4           <<< vertex count of face 5
90
3       <<< face 5, index of 1. vertex
90
7       <<< face 5, index of 2. vertex
90
6       <<< face 5, index of 3. vertex
90
2       <<< face 5, index of 4. vertex
90
4           <<< vertex count of face 6
90
0       <<< face 6, index of 1. vertex
90
```

(continues on next page)

(continued from previous page)

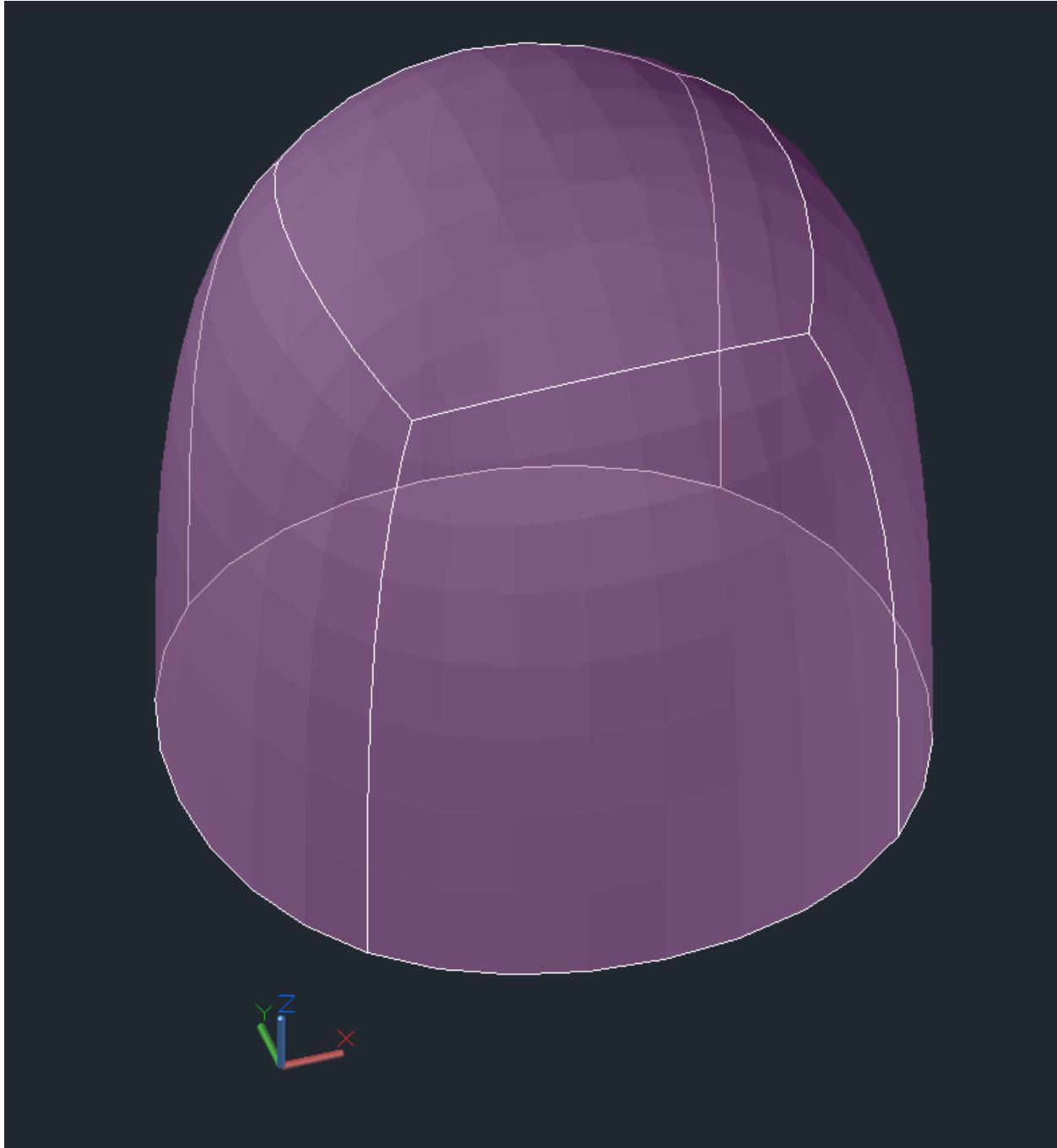
```

4      <<< face 6, index of 2. vertex
90
7      <<< face 6, index of 3. vertex
90
3      <<< face 6, index of 4. vertex
94          <<< edge count, each edge has exact two group code 90 tags
4          <<< the real edge count not the group code 90 tags!
90
0      <<< edge 1, vertex 1
90
1      <<< edge 1, vertex 1
90
1      <<< edge 2, vertex 1
90
2      <<< edge 2, vertex 2
90
2      <<< edge 3, vertex 1
90
3      <<< edge 3, vertex 2
90
3      <<< edge 4, vertex 1
90
0      <<< edge 4, vertex 2
95          <<< edge crease count, has to match edge count!
4
140
3.0    <<< crease value for edge 1
140
3.0    <<< crease value for edge 2
140
3.0    <<< crease value for edge 3
140
3.0    <<< crease value for edge 4
90        <<< property overwrite???
0

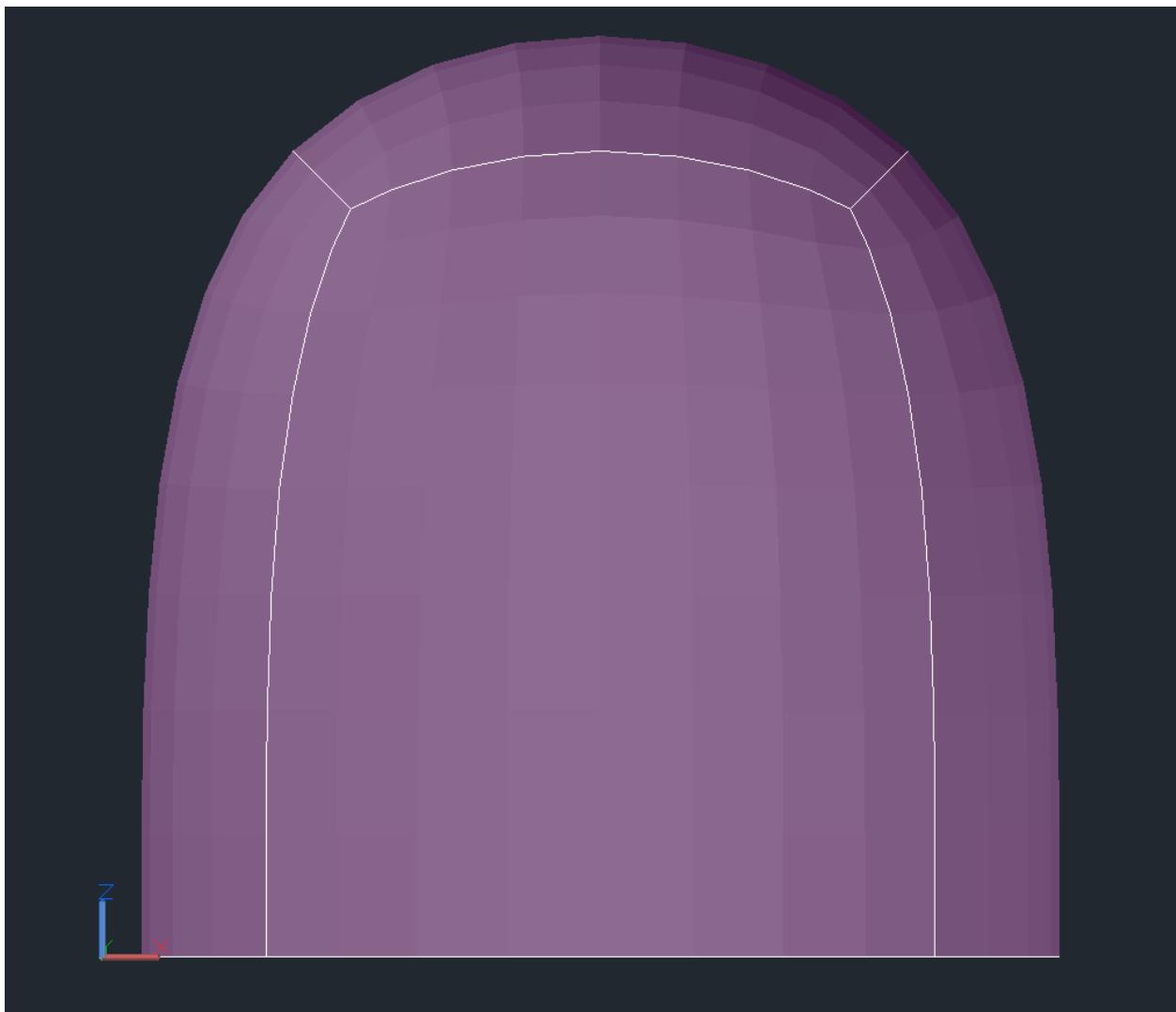
```

The edge and crease data have only a meaning if subdivision of the geometry will be applied! A crease value equal to the subdivision level prevents subdividing for the edge completely, a value between 0.0 and the subdivision level applies subdivision partially.

The cube with subdivision level of 3 and crease values of 3.0:



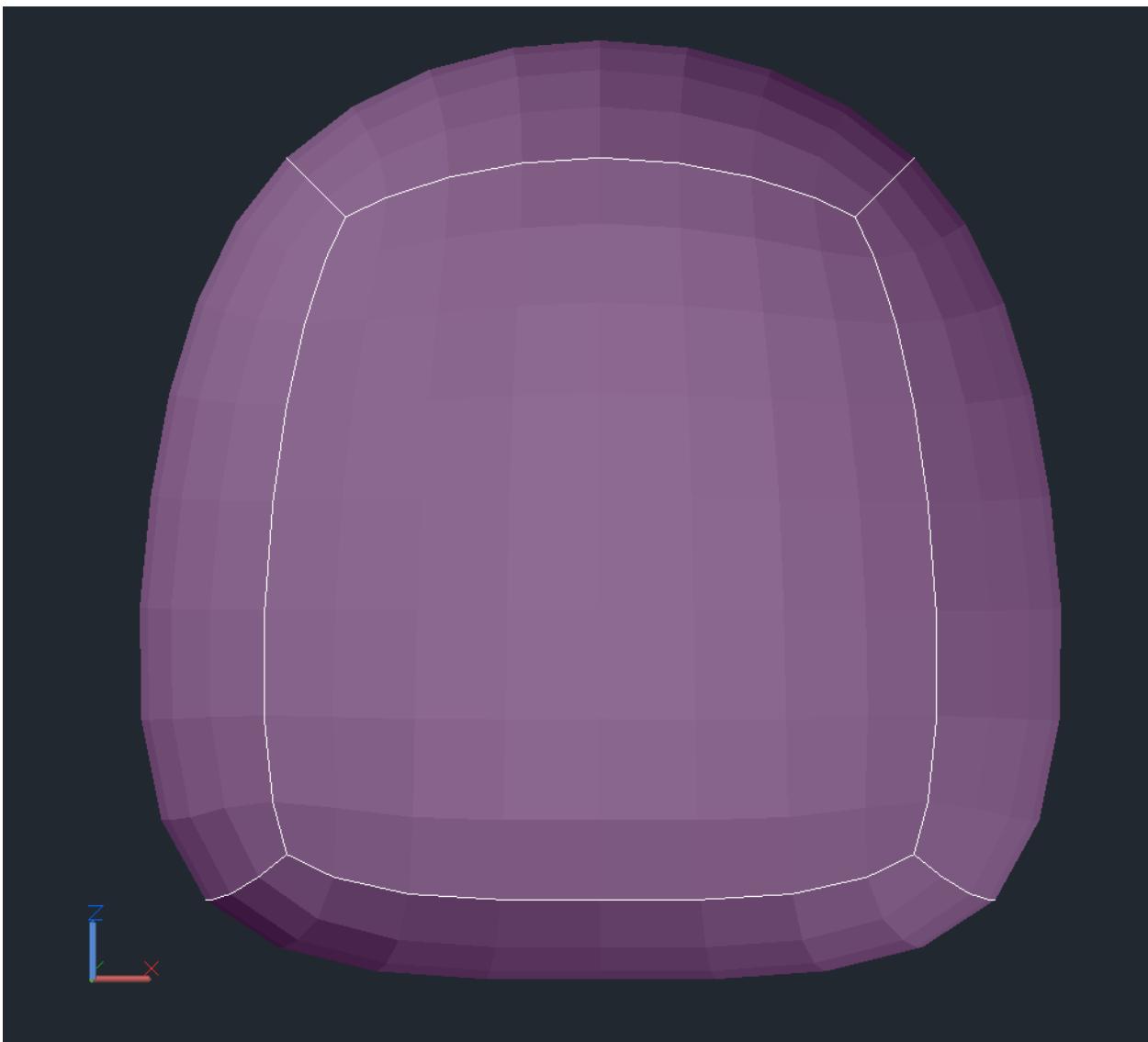
Front view for better details:



The cube with subdivision levels of 3 and crease values of 2.0:



The cube with subdivision level of 3 and crease values of 1.0:



The property overriding protocol is not documented in the DXF reference and currently I have no access to a CAD application which can created property overriding.

MULTILEADER Internals

The MULTILEADER leader is a very complex entity and has also some weird and unique properties.

1. MULTILEADER has the alias name MLEADER which is accepted by any *reliable CAD application*, but all of them create the entity as MULTILEADER
2. uses *raw color* values to define colors
3. creates a complex context data structures beyond simple tags inside the subclass AcDbMLLeader

See also:

- `ezdxf.entities.MultiLeader`
- `ezdxf.entities.MLeaderStyle`

- `ezdxf.render.MultiLeaderBuilder`
- *Tutorial for MultiLeader*
- DXF Reference: [MLEADER](#)

Example for `ezdxf.entities.MLeaderContext` created by BricsCAD:

MTEXT Internals

The MTEXT entity stores multiline text in a single entity and was introduced in DXF version R13/R14. For more information about the top level stuff go to the [MText](#) class.

See also:

- DXF Reference: [MTEXT](#)
- `ezdxf.entities.MText` class

Orientation

The MTEXT entity does not establish an OCS. The entity has a `text_direction` attribute, which defines the local x-axis, the `extrusion` attribute defines the normal vector and the y-axis = extrusion cross x-axis.

The MTEXT entity can have also a `rotation` attribute (in degrees), the x-axis attribute has higher priority than the `rotation` attribute, but it is not clear how to convert the `rotation` attribute into a `text_direction` vector, but for most common cases, where only the `rotation` attribute is present, the `extrusion` is most likely the WCS z-axis and the `rotation` is the direction in the xy-plane.

Text Content

The content text is divided across multiple tags of group code 3 and 1, the last line has the group code 1, each line can have a maximum line length of 255 bytes, but BricsCAD (and AutoCAD?) store only 249 bytes in single line and one byte is not always one char.

Inline Code Specials

The text formatting is done by inline codes, see the [MText](#) class.

Information gathered by implementing the `MTextEditor` and the `MTextParser` classes:

- **caret encoded characters:**
 - “^I” tabulator
 - “^J” (LF) is a valid line break like “\P”
 - “^M” (CR) is ignored
 - other characters render as empty square “”
 - a space “ ” after the caret renders the caret glyph: “1^ 2” renders “1^2”
- **special encoded characters:**
 - “%%c” and “%%C” renders “Ø” (alt-0216)
 - “%%d” and “%%D” renders “ø” (alt-0176)

- “%%p” and “%%P” renders “±” (alt-0177)
- **Alignment command “A”:** argument “0”, “1” or “2” is expected
 - the terminator symbol “;” is optional
 - the arguments “3”, “4”, “5”, “6”, “7”, “8”, “9” and “-” default to 0
 - other characters terminate the command and will be printed: “\AX”, renders “X”
- **ACI color command “C”:** int argument is expected
 - the terminator symbol “;” is optional
 - a leading “-” or “+” terminates the command, “\C+5” renders “\C+5”
 - arguments > 255, are ignored but consumed “\C1000” renders nothing, not even a “0”
 - a trailing “;” after integers is always consumed, even for much to big values, “\C10000;” renders nothing
- **RGB color command “c”:** int argument is expected
 - the terminator symbol “;” is optional
 - a leading “-” or “+” terminates the command, “\c+255” renders “\c+255”
 - arguments ≥ 16777216 are masked by: value & 0xFFFFFFF
 - a trailing “;” after integers is always consumed, even for much to big values, “\c9999999999;” renders nothing and switches the color to yellow (255, 227, 11)
- **Height command “H” and “H...x”:** float argument is expected
 - the terminator symbol “;” is optional
 - a leading “-” is valid, but negative values are ignored
 - a leading “+” is valid
 - a leading “.” is valid like “H.5x” for height factor 0.5
 - exponential format is valid like “H1e2” for height factor 100 and “H1e-2” for 0.01
 - an invalid floating point value terminates the command, “\H1..5” renders “\H1..5”
- **Other commands with floating point arguments like the height command:**
 - Width commands “W” and “W...x”
 - Character tracking commands “\T” and “\T...x”, negative values are used
 - Slanting (oblique) command “Q”
- **Stacking command “S”:**
 - build fractions: “numerator (upr)” + “stacking type char (t)” + “denominator (lwr)” + “;”
 - divider chars: “^”, “/” or “#”
 - a space ” ” after the divider char “^” is mandatory to avoid caret decoding: “\S1^ 2;”
 - the terminator symbol “;” is mandatory to end the command, all chars beyond the “\S” until the next “;” or the end of the string are part of the fraction
 - backslash escape “\;” to render the terminator char
 - a space ” ” after the divider chars “/” and “#” is rendered as space ” ” in front of the denominator
 - the numerator and denominator can contain spaces

- backslashes “\” inside the stacking command are ignored (except “\\;”) “\\$N^ \\$P” render “N” over “P”, therefore property changes (color, text height, ...) are not possible inside the stacking command
 - grouping chars “{” and “}” render as simple curly braces
 - caret encoded chars are decoded “^I”, “^J”, “^M”, but render as a simple space “ ” or as the replacement char “” plus a space
 - a divider char after the first divider char, renders as the char itself: “\\$1/2/3” renders the horizontal fraction “1” / “2/3”
- **Font command “\f” and “\F”: export only “\f”, parse both, “\F” ignores some arguments**
 - the terminator symbol “;” is mandatory to end the command, all chars beyond the “\f” until the next “;” or the end of the string are part of the command
 - the command arguments are separated by the pipe char “|”
 - arguments: “font family name” | “bold” | “italic” | “codepage” | “pitch”; example “\fArialb0li0lc0lp0;”
 - only the “font family name” argument is required, fonts which are not available on the system are replaced by the “TXT.SHX” shape font
 - the “font family name” is the font name shown in font selection widgets in desktop applications
 - “b1” to use the bold font style, any other second char is interpreted as “non bold”
 - “i1” to use an italic font style, any other second char is interpreted as “non italic”
 - “c???” change codepage, “c0” use the default codepage, because of the age of unicode no further investigations, also seems to be ignored by AutoCAD and BricsCAD
 - “p???” change pitch size, “p0” means don’t change, ignored by AutoCAD and BricsCAD, to change the text height use the “H” command
 - the order is not important, but export always in the shown order: “\fArialb0li0;” the arguments “c0” and “p0” are not required

- **Paragraph properties command “\p”**

- the terminator symbol “;” is mandatory to end the command, all chars beyond the “\p” until the next “;” or the end of the string are part of the command
- the command arguments are separated by commas “,”
- all values are factors for the initial char height of the MTEXT entity, example: char height = 2.5, “\pl1;” set the left paragraph indentation to $1 \times 2.5 = 2.5$ drawing units.
- all values are floating point values, see height command
- arguments are “i”, “l”, “r”, “q”, “t”
- a “*” as argument value, resets the argument to the initial value: “i0”, “l0”, “r0”, the “q” argument most likely depends on the text direction; I haven’t seen “t*”. The sequence used by BricsCAD to reset all values is “\pi*, l*, r*, q*, t;”
- “i” indentation of the first line relative to the “l” argument as floating point value, “\pi1.5”
- “l” left paragraph indentation as floating point value, “\pl1.5”
- “r” right paragraph indentation as floating point value, “\pr1.5”
- “x” is required if a “q” or a “t” argument is present, the placement of the “x” has no obvious rules
- “q” paragraph alignment
 - * “ql” left paragraph alignment

- * “qr” right paragraph alignment
- * “qc” center paragraph alignment
- * “qj” justified paragraph alignment
- * “qd” distributed paragraph alignment
- “t” tabulator stops as comma separated list, the default tabulator stops are located at 4, 8, 12, ..., by defining at least one tabulator stop, the default tabulator stops will be ignored. There are 3 kinds of tabulator stops: left, right and center adjusted stops, e.g. “pxt1,r5,c8”:
 - * a left adjusted stop has no leading char, two left adjusted stops “\pxt1,2;”
 - * a right adjusted stop has a preceding “r” char, “\pxtr1,r2;”
 - * a center adjusted stop has a preceding “c” char, “\pxtc1,c2;”
- complex example to create a numbered list with two items: “pxi-3,14t4;1.^Ifirst item\P2.^Isecond item”
- a parser should be very flexible, I have seen several different orders of the arguments and placing the sometimes required “x” has no obvious rules.
- exporting seems to be safe to follow these three rules:
 1. the command starts with “\px”, the “x” does no harm, if not required
 2. argument order “i”, “l”, “r”, “q”, “t”, any of the arguments can be left off
 3. terminate the command with a “;”

Height Calculation

There is no reliable way to calculate the MTEXT height from the existing DXF attributes. The `rect_height` (group code 43) attribute is not required and seldom present. DXF R2007 introduced the `defined_height` attribute to store the defined column height of the MTEXT entity but only in column mode. MTEXT entities without columns, except MTEXT entities created with column type “No Columns”, store always 0.0 as defined column height. Which seems to mean: defined by the rendered text content.

The only way to calculate the MTEXT height is to replicate the rendering results of AutoCAD/BricsCAD by implementing a rendering engine for MTEXT.

In column mode the MTEXT height is stored for every column for DXF version before R2018. In DXF R2018+ the column heights are only stored if `MTextColumns.auto_height` is `False`. If `MTextColumns.auto_height` is `True`. But DXF R2018+ stores the MTEXT total width and height in explicit attributes.

Width Calculation

The situation for width calculation is better than for the height calculation, but the attributes `width` and `rect_width` are not mandatory.

There is a difference between MTEXT entities with and without columns:

Without columns the attribute `width` (reference column width) contains the true entity width if present. A long word can overshoot this width! The `rect_width` attribute is seldom present.

For MTEXT with columns, the `width` attribute is maybe wrong, the correct width for a column is stored in the `column_width` attribute and the `total_width` attribute stores the total width of the MTEXT entity overall columns, see also following section “Column Support”.

Background Filling

The background fill support is available for DXF R2007+. The group code 90 defines the kind of background fill:

0	off
1	color defined by group code 63, 421 or 431
2	drawing window color
3	background (canvas) color
16	bit-flag text frame, see Open Design Alliance Specification 20.4.46

Group codes to define background fill attributes:

45	scaling factor for the border around the text, the value should be in the range of [1, 5], where 1 fits exact the MText entity
63	set the background color by <i>ACI</i> .
421	set the background color as <i>true color</i> value.
431	set the background color by color name - no idea how this works
441	set the transparency of the background fill, not supported by AutoCAD or BricsCAD.

Group codes 45, 90 and 63 are required together if one of them is used. The group code 421 and 431 also requires the group code 63, even this value is ignored.

```
... <snip>
1 <str> eu feugiat nulla facilisis at vero eros et accumsan et iusto ...
73 <int> 1
44 <float> 1.0
90 <int> 1, b00000001    <<< use a color
63 <int> 1                <<< ACI color (red)
45 <float> 1.5            <<< bg scaling factor, relative to the char height
441 <int> 0               <<< ignored (optional)
... <snip>
```

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet,



The background scaling does not alter the `width`, `column_width` or `total_width` attributes. The background acquires additional space around the MTEXT entity.

Columns with background color:

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accusam et iusto odio dignissim qui blandit praesent

luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consetetur adipisciing elit, sed diam nonumy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accusam et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consetetur adipisciing elit, sed diam nonumy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, At accusam aliquyam nonumy sed diam dolore dolores duis at vero eos erat, et nonumy sed diam et et invidunt justo labore et dolore magna aliqua. Kaumno rebum. sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur

Text Frame

The MTEXT entity can have a text frame only, without a background filling, group code 90 has value 16. In this case all other background related tags are removed (45, 63, 421, 431, 441) and the scaling factor is 1.5 by default.

XDATA for Text Frame

This XDATA exist only if the text frame flag in group code 90 is set and for DXF version < R2018!

```
... <snip>
1001 <ctrl> ACAD
1000 <str> ACAD_MTEXT_TEXT_BORDERS_BEGIN
1070 <int> 80      <<< group code for repeated flags
1070 <int> 16      <<< repeated group code 90?
1070 <int> 46      <<< group code for scaling factor, which is fixed?
1040 <float> 1.5    <<< scaling factor
1070 <int> 81      <<< group code for repeated flow direction?
1070 <int> 1        <<< flow direction?
1070 <int> 5        <<< group code for a handle, multiple entries possible
1005 <hex> #A8      <<< handle to the LWPOLYLINE text frame
1070 <int> 5        <<< group code for next handle
1005 <hex> #A9      <<< next handle
...
1000 <str> ACAD_MTEXT_TEXT_BORDERS_END
```

Extra LWPOLYLINE Entity as Text Frame

The newer versions of AutoCAD and BricsCAD get all the information they need from the MTEXT entity, but it seems that older versions could not handle the text frame property correct. Therefore AutoCAD and BricsCAD create a separated LWPOLYLINE entity for the text frame for DXF versions < R2018. The handle to this text frame entity is stored in the XDATA as group code 1005, see section above.

Because this LWPOLYLINE is not required *ezdxf* does **not** create such a text frame entity nor the associated XDATA and *ezdxf* also **removes** this data from loaded DXF files at the second loading stage.

Column Support

CAD applications build multiple columns by linking 2 or more MTEXT entities together. In this case each column is a self-sufficient entity in DXF version R13 until R2013. The additional columns specifications are stored in the XDATA if the MTEXT which represents the first column.

DXF R2018 changed the implementation into a single MTEXT entity which contains all the content text at once and stores the column specification in an embedded object.

Hint: The `width` attribute for the linked MTEXT entities could be wrong. Always use the `column_width` and the `total_width` attributes in column mode.

There are two column types, the **static** type has the same column height for all columns, the **dynamic** type can have the same (auto) height or an individual height for each column.

Common facts about columns for all column types:

- all columns have the same column width

- all columns have the same gutter width
- the top of the column are at the same height

Column Type

The column type defines how a CAD application should create the columns, this is not important for the file format, because the result of this calculation, the column count and the column height, is stored the DXF file.

Column Type in BricsCAD	Description
Static	All columns have the same height. The “auto height” flag is 0.
Dynamic (auto height)	Same as the static type, all columns have the same height. The “auto height” flag is 1. The difference to the static type is only important for interactive CAD applications.
Dynamic (manual height)	same as the dynamic (auto height) type, but each column can have an individual height.
No column	A regular MTEXT with “defined column height” attribute?

Column Type	Defined Height	Auto Height	Column Heights
Static	stored	False	not stored
Dynamic auto	stored	True	not stored
Dynamic manual	not stored	False	stored (last=0)

Column Count

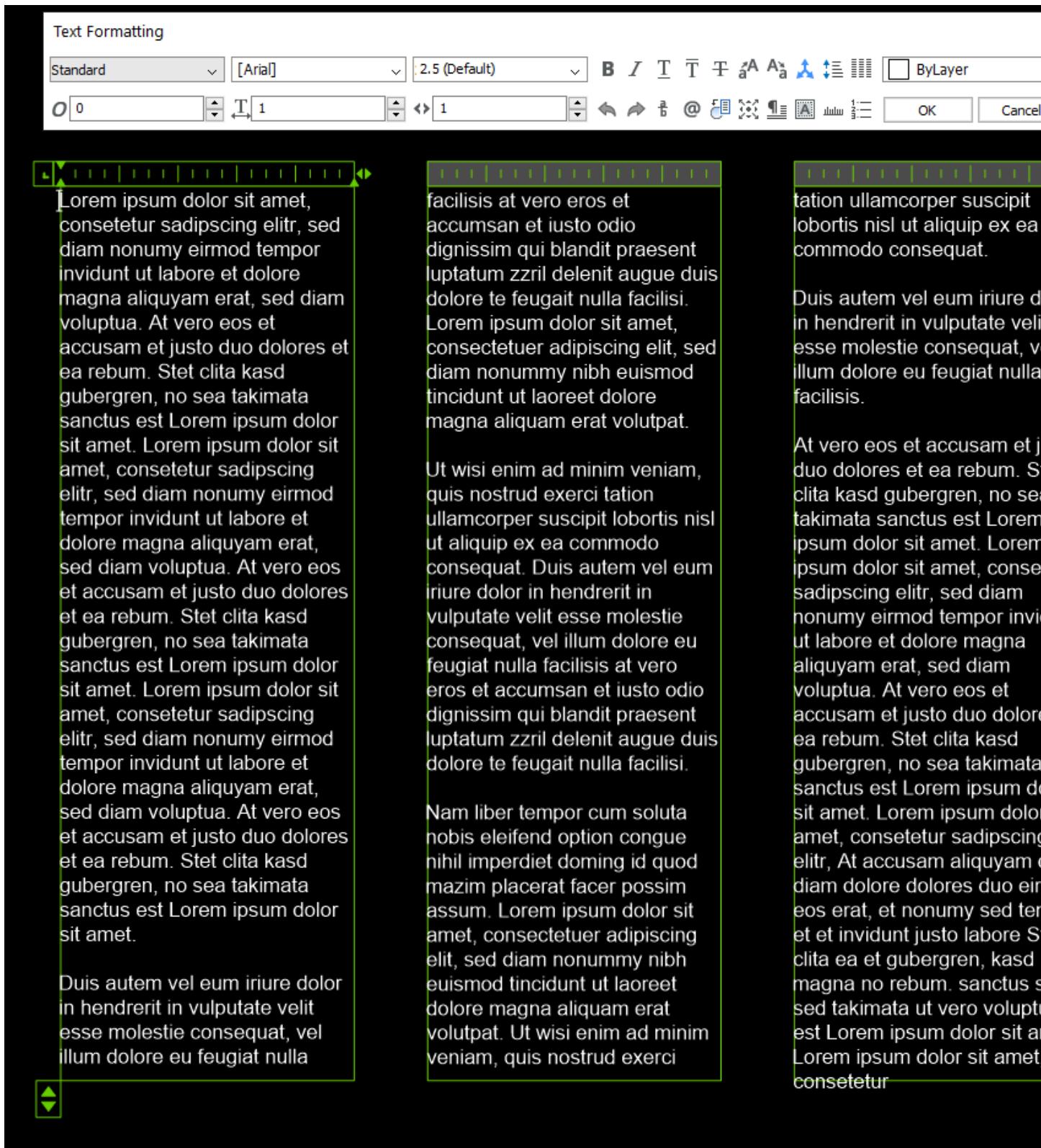
For DXF versions < R2018 the column count is always given by the count of linked MTEXT columns. Caution: the column count stored in the XDATA section by group code 76 may not match the count of linked MTEXT entities and AutoCAD is OK with that! In DXF R2018+ this property is not available, because there are no linked MTEXT entities anymore.

R2018+: For the column types “static” and “dynamic manual” the correct column count is stored as group code 72. For the column type “dynamic auto” the stored *column count is 0*. It is possible to calculate the column count from the total width and the column width if the total width is correct like in AutoCAD and BricsCAD.

Static Columns R2000

Example for a **static** column specification:

- Column Type: Static
- Number of Columns: 3
- Height: 150.0, manual entered value and all columns have the same height
- Width: 50.0
- Gutter Width: 12.5



The column height is stored as the “defined column height” in XDATA (46) or the embedded object (41).

DXF R2000 example with a static column specification stored in XDATA:

```
0
MTEXT
5           <<< entity handle
9D
102
{ACAD_XDICTIONARY
360
9F
102
}
330           <<< block record handle of owner layout
1F
100
AcDbEntity
8           <<< layer
0
100           <<< begin of MTEXT specific data
AcDbMText
10           <<< (10, 20, 30) insert location in WCS
285.917876152751
20
276.101821192053
30
0.0
40           <<< character height in drawing units
2.5
41           <<< reference column width, if not in column mode
62.694...   <<< in column mode: the real column is defined in XDATA (48)
71           <<< attachment point
1
72           <<< text flow direction
1
3           <<< begin of text
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam ...
3
kimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit ...
3
ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ...
3
At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd ...
3
ore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio ...
1           <<< last text line and end of text
euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.
73           <<< line spacing style
1
44           <<< line spacing factor
1.0
1001
AcadAnnotative
1000
AnnotativeData
1002
{
1070
1
1070
```

(continues on next page)

(continued from previous page)

```

0
1002
}
1001      <<< AppID "ACAD" contains the column specification
ACAD
1000
ACAD_MTEXT_COLUMN_INFO_BEGIN
1070
75      <<< group code column type
1070
1      <<< column type: 0=no column; 1=static columns; 2=dynamic columns
1070
79      <<< group code column auto height
1070
0      <<< flag column auto height
1070
76      <<< group code column count
1070
3      <<< column count
1070
78      <<< group code column flow reversed
1070
0      <<< flag column flow reversed
1070
48      <<< group code column width
1040
50.0     <<< column width in column mode
1070
49      <<< group code column gutter
1040
12.5     <<< column gutter width
1000
ACAD_MTEXT_COLUMN_INFO_END
1000      <<< linked MTEXT entities specification
ACAD_MTEXT_COLUMNS_BEGIN
1070
47      <<< group code for column count, incl. the 1st column - this entity
1070
3      <<< column count
1005
1B4      <<< handle to 2nd column as MTEXT entity
1005
1B5      <<< handle to 3rd column as MTEXT entity
1000
ACAD_MTEXT_COLUMNS_END
1000
ACAD_MTEXT_DEFINED_HEIGHT_BEGIN
1070
46      <<< group code for defined column height
1040
150.0     <<< defined column height
1000
ACAD_MTEXT_DEFINED_HEIGHT_END

```

The linked column MTEXT #1B4 in a compressed representation:

```
0 <ctrl> MTEXT
...
100 <ctrl> AcDbMText
10 <point> (348.417876152751, 276.101821192053, 0.0)
40 <float> 2.5
41 <float> 175.0      <<< invalid reference column width
...
1001 <ctrl> ACAD
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_BEGIN
1070 <int> 46          <<< defined column height
1040 <float> 150.0
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_END
```

The linked MTEXT has no column specification except the “defined column height” in the XDATA. The reference column width is not the real value of 50.0, see XDATA group code 48 in the main MTEXT #9D, instead the total width of 175.0 is stored at group code 41. This is problem if a renderer try to render this MTEXT as a standalone entity. The renderer has to fit the content into the column width by itself and without the correct column width, this will produce an incorrect result.

There exist no back link to the main MTEXT #9D. The linked MTEXT entities appear after the main MTEXT in the layout space, but there can be other entities located between these linked MTEXT entities.

The linked column MTEXT #1B5:

```
0 <ctrl> MTEXT
5 <hex> #1B5
...
100 <ctrl> AcDbMText
10 <point> (410.917876152751, 276.101821192053, 0.0)
40 <float> 2.5
41 <float> 175.0      <<< invalid reference column width
...
1001 <ctrl> ACAD
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_BEGIN
1070 <int> 46          <<< defined column height
1040 <float> 150.0
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_END
```

Static Columns R2018

The MTEXT entity in DXF R2018 contains all column information in a single entity. The text content of all three columns are stored in a continuous text string, the separation into columns has to be done by the renderer. The manual column break \N is **not** used to indicate automatic column breaks. The MTEXT renderer has to replicate the AutoCAD/BricsCAD rendering as exact as possible to achieve the same results, which is very hard without rendering guidelines or specifications.

The example from above in DXF R2018 with a static column specification stored in an embedded object:

```
0
MTEXT
5      <<< entity handle
9D
102
{ACAD_XDICTIONARY
360
9F
```

(continues on next page)

(continued from previous page)

```

102
}
330      <<< block record handle of owner layout
1F
100
AcDbEntity
8       <<< layer
0
100
AcDbMText
10      <<< (10, 20, 30) insert location in WCS
285.917876152751
20
276.101821192053
30
0.0
40      <<< character height in drawing units
2.5
41      <<< reference column width, if not in column mode
62.694536423841
46      <<< defined column height
150.0
71      <<< attachment point
1
72      <<< text flow direction
1
3      <<< text content of all three columns
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam n...
3
imata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit...
3
a rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lor...
3
vero eos et accusam et justo duo dolores et ea rebum. Stet clita ka...
3
eu feugiat nulla facilisis at vero eros et accumsan et iusto odio s...
3
od tincidunt ut laboret dolore magna aliquam erat volutpat. \P\PU...
3
e velit esse molestie consequat, vel illum dolore eu feugiat nulla ...
3
obis eleifend option congue nihil imperdiet doming id quod mazim pl...
3
m ad minim veniam, quis nostrud exerci tation ullamcorper suscipit ...
3
lisis. \P\PAAt vero eos et accusam et justo duo dolores et ea rebu...
3
t laboret dolore magna aliquyam erat, sed diam voluptua. At vero ...
3
litr, At accusam aliquyam diam diam dolore dolores duo eirmod eos e...
1
ipsum dolor sit amet, consetetur
73      <<< line spacing style
1
44      <<< line spacing factor
1.0
101     <<< column specification as embedded object

```

(continues on next page)

(continued from previous page)

```
Embedded Object
70      <<< ???
1
10      <<< (10, 20, 30) text direction vector (local x-axis)
1.0
20
0.0
30
0.0
11      <<< (11, 21, 31) repeated insert location of AcDbMText
285.917876152751
21
276.101821192053
31
0.0
40      <<< repeated reference column width
62.694536423841
41      <<< repeated defined column height
150.0
42      <<< extents (total) width
175.0
43      <<< extents (total) height, max. height if different column heights
150.0
71      <<< column type: 0=no column; 1=static columns; 2=dynamic columns
1
72      <<< column height count
3
44      <<< column width
50.0
45      <<< column gutter width
12.5
73      <<< flag column auto height
0
74      <<< flag reversed column flow
0
1001
AcadAnnotative
1000
AnnotativeData
1002
{
1070
1
1070
0
1002
}
```

Dynamic (auto height) Columns R2000

Example for a **dynamic** column specification:

- Column Type: Dynamic
- Number of Columns: 3
- Height: 158.189... adjusted by widget and all columns have the same height

- Width: 50.0
- Gutter Width: 12.5

```

0 <ctrl> MTEXT
5 <hex> #A2           <<< entity handle
... <snip>
330 <hex> #1F           <<< block record handle of owner layout
100 <ctrl> AcDbEntity
8 <str> 0           <<< layer
100 <ctrl> AcDbMText
10 <point> (-133.714579865783, 276.101821192053, 0.0)   <<< insert location in WCS
40 <float> 2.5          <<< character height in drawing units
41 <float> 62.694536423841 <<< reference column width, if not in column mode
71 <int> 1           <<< attachment point
72 <int> 1           <<< flag text flow direction
3 <str> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed dia...
... <snip>
73 <int> 1           <<< line spacing style
44 <float> 1.0          <<< line spacing factor
1001 <ctrl> AcadAnnotative
... <snip>
1001 <ctrl> ACAD
1000 <str> ACAD_MTEXT_COLUMN_INFO_BEGIN
1070 <int> 75          <<< column type: 2=dynamic columns
1070 <int> 2
1070 <int> 79          <<< flag column auto height
1070 <int> 1
1070 <int> 76          <<< column count
1070 <int> 3
1070 <int> 78          <<< flag column flow reversed
1070 <int> 0
1070 <int> 48          <<< column width in column mode
1040 <float> 50.0
1070 <int> 49          <<< column gutter width
1040 <float> 12.5
1000 <str> ACAD_MTEXT_COLUMN_INFO_END
1000 <str> ACAD_MTEXT_COLUMNS_BEGIN
1070 <int> 47          <<< column count
1070 <int> 3
1005 <hex> #1B6          <<< handle to 2. column as MTEXT entity
1005 <hex> #1B7          <<< handle to 3. column as MTEXT entity
1000 <str> ACAD_MTEXT_COLUMNS_END
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_BEGIN
1070 <int> 46          <<< defined column height
1040 <float> 158.189308131867
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_END

```

The linked column MTEXT #1B6:

```

0 <ctrl> MTEXT
... <snip>
100 <ctrl> AcDbMText
10 <point> (-71.214579865783, 276.101821192053, 0.0)
40 <float> 2.5
41 <float> 175.0    <<< invalid column width
... <snip>
1001 <ctrl> ACAD

```

(continues on next page)

(continued from previous page)

```

1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_BEGIN
1070 <int> 46      <<< defined column height
1040 <float> 158.189308131867
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_END

```

The linked column MTEXT #1B7:

```

0 <ctrl> MTEXT
... <snip>
100 <ctrl> AcDbMText
10 <point> (-8.714579865783, 276.101821192053, 0.0)
40 <float> 2.5
41 <float> 175.0    <<< invalid column width
... <snip>
1001 <ctrl> ACAD
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_BEGIN
1070 <int> 46      <<< defined column height
1040 <float> 158.189308131867
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_END

```

Dynamic (auto height) Columns R2018

```

0 <ctrl> MTEXT
5 <hex> #A2          <<< entity handle
102 <ctrl> {ACAD_XDICTIONARY
360 <hex> #A3
102 <ctrl> }
330 <hex> #1F          <<< block record handle of owner layout
100 <ctrl> AcDbEntity
8 <str> 0            <<< layer
100 <ctrl> AcDbMText
10 <point> (-133.714579865783, 276.101821192053, 0.0)  <<< insert location in WCS
40 <float> 2.5        <<< character height in drawing units
41 <float> 62.694536423841   <<< reference column width, if not in column mode
46 <float> 158.189308131867  <<< defined column height
71 <int> 1            <<< attachment point
72 <int> 1            <<< text flow direction
3 <str> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam...
... <snip> text content of all three columns
73 <int> 1            <<< line spacing style
44 <float> 1.0         <<< line spacing factor
101 <ctrl> Embedded Object
70 <int> 1, b00000001  <<< ???
10 <point> (1.0, 0.0, 0.0)  <<< text direction vector (local x-axis)
11 <point> (-133.714579865783, 276.101821192053, 0.0)  <<< repeated insert location
40 <float> 62.694536423841  <<< repeated reference column width
41 <float> 158.189308131867  <<< repeated defined column height
42 <float> 175.0        <<< extents (total) width
43 <float> 158.189308131867  <<< extents (total) height, max. height if different
→column heights
71 <int> 2            <<< column type: 2=dynamic columns
72 <int> 0            <<< column height count
44 <float> 50.0        <<< column width
45 <float> 12.5        <<< column gutter width

```

(continues on next page)

(continued from previous page)

```

73 <int> 1           <<< flag column auto height
74 <int> 0           <<< flag reversed column flow
1001 <ctrl> AcadAnnotative
1000 <str> AnnotativeData
1002 <str> {
1070 <int> 1
1070 <int> 0
1002 <str> }

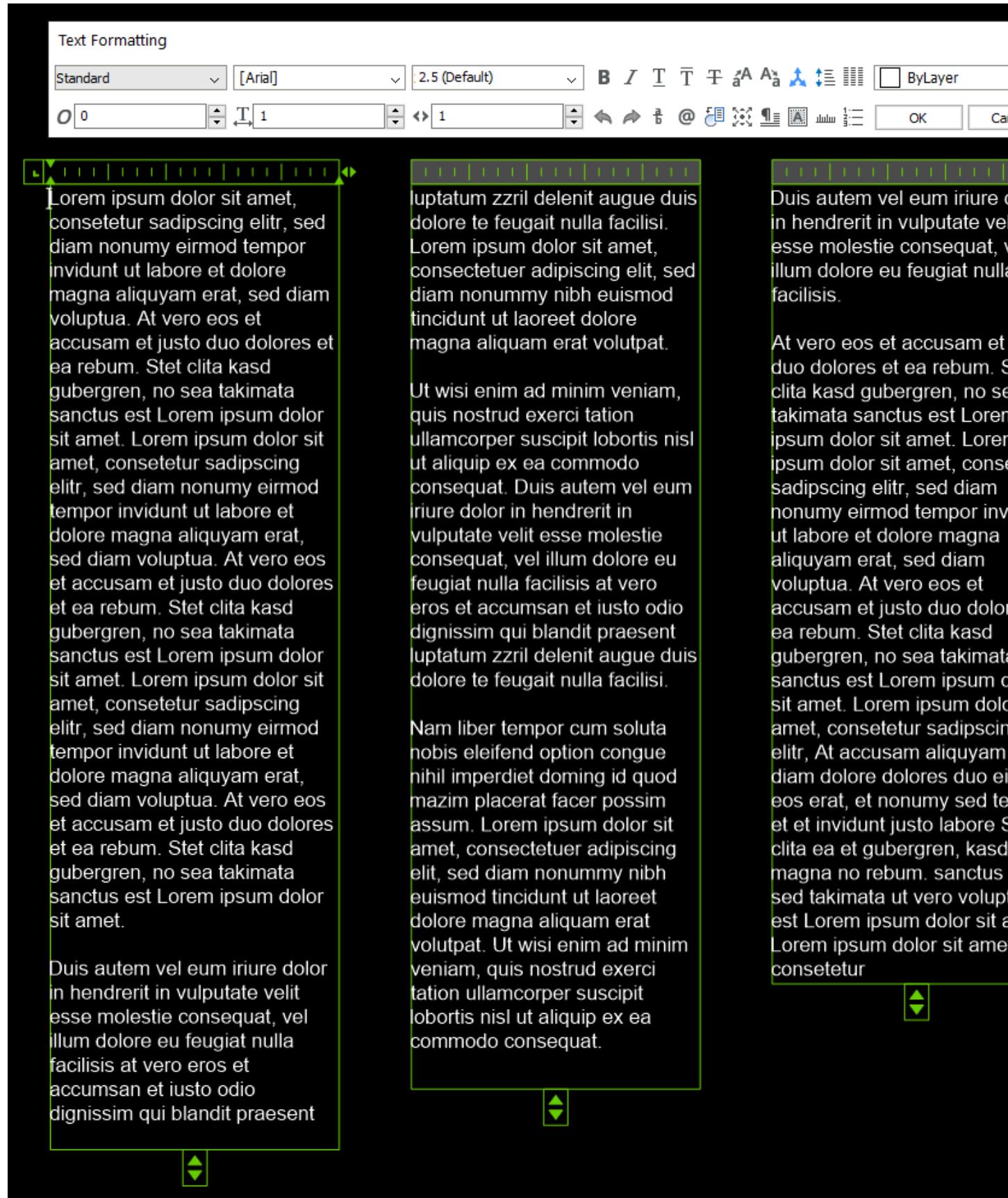
```

Dynamic (manual height) Columns R2000

Example for a **dynamic** column specification with manual height definition for three columns with different column heights. None of the (linked) MTEXT entities does contain XDATA for the defined column height.

Hint: If “content type” is 2 and flag “column auto height” is 0, no defined height in XDATA.

- Column Type: Dynamic
- Number of Columns: 3
- Height: 164.802450331126, max. column height
- Width: 50.0
- Gutter Width: 12.5



```

0 <ctrl> MTEXT
5 <hex> #9C                         <<< entity handle
330 <hex> #1F                         <<< block record handle of owner layout
100 <ctrl> AcDbEntity
8 <str> 0                             <<< layer
100 <ctrl> AcDbMText
10 <point> (69.806121185863, 276.101821192053, 0.0)    <<< insert location in WCS
40 <float> 2.5                         <<< character height in drawing units
41 <float> 62.694536423841            <<< reference column width, if not in column mode
71 <int> 1                            <<< attachment point
72 <int> 1                            <<< flag text flow direction
3 <str> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, ...
... <snip>
73 <int> 1                            <<< line spacing style
44 <float> 1.0                         <<< line spacing factor
1001 <ctrl> ACAD
1000 <str> ACAD_MTEXT_COLUMN_INFO_BEGIN
1070 <int> 75                          <<< column type: 2=dynamic columns
1070 <int> 2                           <<< flag column auto height
1070 <int> 79                          <<< column count
1070 <int> 0                           <<< column count
1070 <int> 76                          <<< flag column flow reversed
1070 <int> 0                           <<< column width in column mode
1070 <int> 48                          <<< column gutter width
1040 <float> 50.0
1070 <int> 49                          <<< column height count
1040 <float> 12.5
1070 <int> 50                          <<< column height count
1070 <int> 3                           <<< column height 1. column
1040 <float> 164.802450331126        <<< column height 2. column
1040 <float> 154.311699779249        <<< column height 3. column, takes the rest?
1000 <str> ACAD_MTEXT_COLUMN_INFO_END
1000 <str> ACAD_MTEXT_COLUMNS_BEGIN
1070 <int> 47                          <<< column count
1070 <int> 3                           <<< handle to 2. column as MTEXT entity
1005 <hex> #1B2
1005 <hex> #1B3                         <<< handle to 3. column as MTEXT entity
1000 <str> ACAD_MTEXT_COLUMNS_END

```

The linked column MTEXT #1B2:

```

0 <ctrl> MTEXT
... <snip>
100 <ctrl> AcDbMText
10 <point> (132.306121185863, 276.101821192053, 0.0)
40 <float> 2.5
41 <float> 175.0                      <<< invalid reference column width
... <snip>
73 <int> 1
44 <float> 1.0

```

The linked column MTEXT #1B3:

```
0 <ctrl> MTEXT
```

(continues on next page)

(continued from previous page)

```

... <snip>
100 <ctrl> AcDbMText
10 <point> (194.806121185863, 276.101821192053, 0.0)
40 <float> 2.5
41 <float> 175.0           <<< invalid reference column width
... <snip>
73 <int> 1
44 <float> 1.0

```

Dynamic (manual height) Columns R2018

Hint: If “content type” is 2 and flag “column auto height” is 0, the “defined column height” is 0.0.

```

0 <ctrl> MTEXT
5 <hex> #9C           <<< entity handle
330 <hex> #1F
100 <ctrl> AcDbEntity
8 <str> 0           <<< block record handle of owner layout
100 <ctrl> AcDbMText
10 <point> (69.806121185863, 276.101821192053, 0.0)    <<< insert location in WCS
40 <float> 2.5           <<< character height in drawing units
41 <float> 62.694536423841      <<< reference column width, if not in column mode
46 <float> 0.0           <<< defined column height
71 <int> 1           <<< attachment point
72 <int> 1           <<< text flow direction
3 <str> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam...
... <snip> text content of all three columns
73 <int> 1           <<< line spacing style
44 <float> 1.0           <<< line spacing factor
101 <ctrl> Embedded Object
70 <int> 1, b00000001     <<< ???
10 <point> (1.0, 0.0, 0.0)    <<< text direction vector (local x-axis)
11 <point> (69.806121185863, 276.101821192053, 0.0)    <<< repeated insert location
40 <float> 62.694536423841      <<< repeated reference column width
41 <float> 0.0           <<< repeated defined column height
42 <float> 175.0           <<< extents (total) width
43 <float> 164.802450331126      <<< extents (total) height, max. height if different
→column heights
71 <int> 2           <<< column type: 2=dynamic columns
72 <int> 3           <<< column height count
44 <float> 50.0           <<< column width
45 <float> 12.5           <<< column gutter width
73 <int> 0           <<< flag column auto height
74 <int> 0           <<< flag reversed column flow
46 <float> 164.802450331126      <<< column height 1. column
46 <float> 154.311699779249      <<< column height 2. column
46 <float> 0.0           <<< column height 3. column, takes the rest?

```

No Columns R2000

I have no idea why this column type exist, but at least provides a reliable value for the MTEXT height by the “defined column height” attribute. The column type is not stored in the MTEXT entity and is therefore not detectable!

- Column Type: No columns
- Number of Columns: 1
- Height: 158.189308131867, defined column height
- Width: 175.0, reference column width

```

0 <ctrl> MTEXT
... <snip>
100 <ctrl> AcDbMText
10 <point> (-344.497343455795, 276.101821192053, 0.0)    <<< insert location in WCS
40 <float> 2.5          <<< character height in drawing units
41 <float> 175.0        <<< reference column width
71 <int> 1              <<< attachment point
72 <int> 1              <<< flag text flow direction
3 <str> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam...
... <snip> text content of all three columns
73 <int> 1              <<< line spacing style
44 <float> 1.0          <<< line spacing factor
... <snip>
1001 <ctrl> ACAD
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_BEGIN
1070 <int> 46           <<< defined column height
1040 <float> 158.189308131867
1000 <str> ACAD_MTEXT_DEFINED_HEIGHT_END

```

No Columns R2018

Does not contain an embedded object.

```

0 <ctrl> MTEXT
... <snip>
100 <ctrl> AcDbMText
10 <point> (-334.691900433414, 276.101821192053, 0.0)    <<< insert location in WCS
40 <float> 2.5          <<< character height in drawing units
41 <float> 175.0        <<< reference column width
46 <float> 158.189308131867  <<< defined column height
71 <int> 1              <<< attachment point
72 <int> 1              <<< flag text flow direction
3 <str> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, ...
... <snip>
73 <int> 1              <<< line spacing style
44 <float> 1.0          <<< line spacing factor
1001 <ctrl> AcadAnnotative
... <snip>

```

DXF Objects

TODO

6.12.3 Management Structures

Block Management Structures

A BLOCK is a layout like the modelspace or a paperspace layout, with the similarity that all these layouts are containers for graphical DXF entities. This block definition can be referenced in other layouts by the INSERT entity. By using block references, the same set of graphical entities can be located multiple times at different layouts, this block references can be stretched and rotated without modifying the original entities. A block is referenced only by its name defined by the DXF tag (2, name), there is a second DXF tag (3, name2) for the block name, which is not further documented by Autodesk, just ignore it.

The (10, base_point) tag (in BLOCK defines a insertion point of the block, by ‘inserting’ a block by the INSERT entity, this point of the block is placed at the location defined by the (10, insert) tag in the INSERT entity, and it is also the base point for stretching and rotation.

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

See also:

- ezdxf DXF Internals: [BLOCKS Section](#)
- DXF Reference: [BLOCKS Section](#)
- DXF Reference: [BLOCK Entity](#)
- DXF Reference: [ENDBLK Entity](#)
- DXF Reference: [INSERT Entity](#)

Block Names

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

Block Definitions in DXF R12

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

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

DXF R12 BLOCKS structure:

```
0      <<< start of a SECTION
SECTION
2      <<< start of BLOCKS section
BLOCKS
...
...      <<< modelspace and paperspace block definitions not shown,
...      <<< see layout management
...
0      <<< start of a BLOCK definition
BLOCK
```

(continues on next page)

(continued from previous page)

```

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

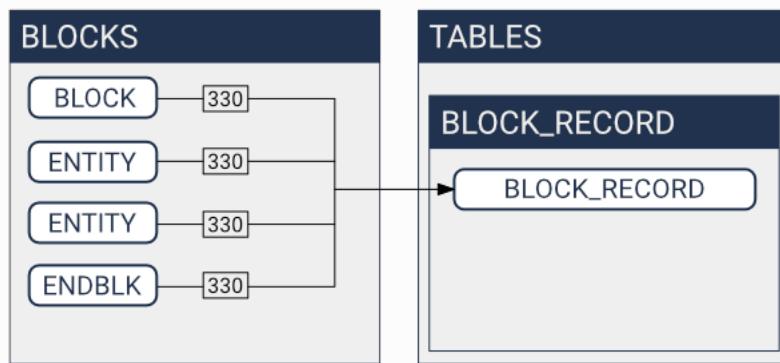
```

Block Definitions in DXF R2000+

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

Especially the concept of ownership is important. Since DXF R13 every graphic entity is associated to a specific layout and a BLOCK definition is also a layout. So all entities in the BLOCK definition, including the BLOCK and the ENDBLK entities, have an owner tag (330, ...), which points to a BLOCK_RECORD entry in the BLOCK_RECORD table. This BLOCK_RECORD is the main management structure for all layouts and is the real owner of the layout entities.

As you can see in the chapter about [Layout Management Structures](#), this concept is also valid for modelspace and paperspace layouts, because these layouts are also BLOCKS, with the special difference, that the entities of the modelspace and the *active* paperspace layout are stored in the ENTITIES section.



See also:

- [DXF R13 and later tag structure](#)
- ezdxf DXF Internals: [TABLES Section](#)
- DXF Reference: [TABLES Section](#)
- DXF Reference: [BLOCK_RECORD Entity](#)

DXF R13 BLOCKS structure:

```

0          <<< start of a SECTION
SECTION
2          <<< start of BLOCKS section
BLOCKS
...
...        <<< modelspace and paperspace block definitions not shown,
...        <<< see layout management
0          <<< start of BLOCK definition
BLOCK
5          <<< even BLOCK gets a handle now ;)
23A
330        <<< owner tag, the owner of a BLOCK is a BLOCK_RECORD in the
...        BLOCK_RECORD table
238
100        <<< subclass marker
AcDbEntity
8          <<< layer of the BLOCK definition
0
100        <<< subclass marker

```

(continues on next page)

(continued from previous page)

```

AcDbBlockBegin
2           <<< BLOCK name
ArchTick
70          <<< flags
0
10         <<< base point, x
0.0
20         <<< base point, y
0.0
30         <<< base point, z
0.0
3           <<< second BLOCK name, same as (2, name)
ArchTick
1           <<< xref name, if block is an external reference
               <<< empty string!
0           <<< start of the first entity of the BLOCK
LWPOLYLINE
5
239
330          <<< owner tag of LWPOLYLINE
238          <<< handle of the BLOCK_RECORD!
100
AcDbEntity
8
0
6
ByBlock
62
0
100
AcDbPolyline
90
2
70
0
43
0.15
10
-0.5
20
-0.5
10
0.5
20
0.5
0           <<< ENDBLK entity, marks the end of the BLOCK definition
ENDBLK
5           <<< handle
23B
330          <<< owner tag, same BLOCK_RECORD as for the BLOCK entity
238
100          <<< subclass marker
AcDbEntity
8           <<< ENDBLK requires the same layer as the BLOCK entity!
0
100          <<< subclass marker
AcDbBlockEnd

```

(continues on next page)

(continued from previous page)

```

0           <<< start of the next BLOCK
BLOCK
...
0
ENDBLK
...
0           <<< end of the BLOCKS section
ENDSEC

```

DXF R13 BLOCK_RECORD structure:

```

0           <<< start of a SECTION
SECTION
2           <<< start of TABLES section
TABLES
0           <<< start of a TABLE
TABLE
2           <<< start of the BLOCK_RECORD table
BLOCK_RECORD
5           <<< handle of the table
1
330          <<< owner tag of the table
0           <<< is always #0
100          <<< subclass marker
AcDbSymbolTable
70          <<< count of table entries, not reliable
4
0           <<< start of first BLOCK_RECORD entry
BLOCK_RECORD
5           <<< handle of BLOCK_RECORD, in ezdxf often referred to as "layout key"
1F
330          <<< owner of the BLOCK_RECORD is the BLOCK_RECORD table
1
100          <<< subclass marker
AcDbSymbolTableRecord
100          <<< subclass marker
AcDbBlockTableRecord
2           <<< name of the BLOCK or LAYOUT
*Model_Space
340          <<< pointer to the associated LAYOUT object
4AF
70           <<< AC1021 (R2007) block insertion units
0
280          <<< AC1021 (R2007) block explodability
1
281          <<< AC1021 (R2007) block scalability
0

...           <<< paperspace not shown
...
0           <<< next BLOCK_RECORD
BLOCK_RECORD
5           <<< handle of BLOCK_RECORD, in ezdxf often referred to as "layout key"
238
330          <<< owner of the BLOCK_RECORD is the BLOCK_RECORD table
1
100          <<< subclass marker

```

(continues on next page)

(continued from previous page)

```

AcDbSymbolTableRecord
100      <<< subclass marker
AcDbBlockTableRecord
2      <<< name of the BLOCK
ArchTick
340      <<< pointer to the associated LAYOUT object
0      <<< #0, because BLOCK doesn't have an associated LAYOUT object
70      <<< AC1021 (R2007) block insertion units
0
280      <<< AC1021 (R2007) block explodability
1
281      <<< AC1021 (R2007) block scalability
0
0      <<< end of BLOCK_RECORD table
ENDTAB
0      <<< next TABLE
TABLE
...
0
ENDTAB
0      <<< end of TABLES section
ENDESC

```

Layout Management Structures

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

1. modelspace contains the ‘real’ world representation of the drawing subjects in real world units.
2. paperspace layouts are used to create different drawing sheets of the modelspace subjects for printing or PDF export
3. Blocks are reusable sets of graphical entities, inserted/referenced by the INSERT entity.

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

See also:

Information about [Block Management Structures](#)

The name of the modelspace BLOCK is “*Model_Space” (DXF R12: “\$MODEL_SPACE”) and the name of the *active* paperspace BLOCK is “*Paper_Space” (DXF R12: “\$PAPER_SPACE”), the entities of these two layouts are stored in the ENTITIES section, DXF R12 supports just one paperspace layout.

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

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

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

LAYOUT Entity

Since DXF R2000 modelspace and paperspace layouts require the DXF LAYOUT entity.

```
0
LAYOUT
5      <<< handle
59
102     <<< extension dictionary (ignore)
{ACAD_XDICTIONARY
360
1C3
102
}
102     <<< reactor (required?)
{ACAD.REACTORS
330
1A     <<< pointer to "ACAD_LAYOUT" DICTIONARY (layout management table)
102
}
330     <<< owner handle
1A     <<< pointer to "ACAD_LAYOUT" DICTIONARY (same as reactor pointer)
100     <<< PLOTSETTINGS
AcDbPlotSettings
1     <<< page setup name

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

40    <<< size of unprintable margin on left side of paper in millimeters,
...    defines also the plot origin-x
6.35
41    <<< size of unprintable margin on bottom of paper in millimeters,
...    defines also the plot origin-y
6.35
42    <<< size of unprintable margin on right side of paper in millimeters
6.35
43    <<< size of unprintable margin on top of paper in millimeters
6.35
44    <<< plot paper size: physical paper width in millimeters
215.90
45    <<< plot paper size: physical paper height in millimeters
279.40
46    <<< X value of plot origin offset in millimeters, moves the plot origin-x
0.0
47    <<< Y value of plot origin offset in millimeters, moves the plot origin-y
0.0
```

(continues on next page)

(continued from previous page)

```

48      <<< plot window area: X value of lower-left window corner
0.0
49      <<< plot window area: Y value of lower-left window corner
0.0
140     <<< plot window area: X value of upper-right window corner
0.0
141     <<< plot window area: Y value of upper-right window corner
0.0
142     <<< numerator of custom print scale: real world (paper) units, 1.0
...
for scale 1:50
1.0
143     <<< denominator of custom print scale: drawing units, 50.0
...
for scale 1:50
1.0
70      <<< plot layout flags, bit-coded (... too many options)
688      <<< b1010110000 = UseStandardScale(16)/PlotPlotStyle(32)
...
PrintLineweights(128)/DrawViewportsFirst(512)
72      <<< plot paper units (0/1/2 for inches/millimeters/pixels), are
...
pixels really supported?
0
73      <<< plot rotation (0/1/2/3 for 0deg/90deg counter-cw/upside-down/90deg cw)
1      <<< 90deg clockwise
74      <<< plot type 0-5 (... too many options)
5      <<< 5 = layout information
7      <<< current plot style name, e.g. 'acad.ctb' or 'acadlt.ctb'

75      <<< standard scale type 0-31 (... too many options)
16     <<< 16 = 1:1, also 16 if user scale type is used
147     <<< unit conversion factor
1.0     <<< for plot paper units in mm, else 0.03937... (1/25.4) for inches
...
as plot paper units
76     <<< shade plot mode (0/1/2/3 for as displayed/wireframe/hidden/rendered)
0     <<< as displayed
77     <<< shade plot resolution level 1-5 (... too many options)
2     <<< normal
78     <<< shade plot custom DPI: 100-32767, Only applied when shade plot
...
resolution level is set to 5 (Custom)
300
148     <<< paper image origin: X value
0.0
149     <<< paper image origin: Y value
0.0
100    <<< LAYOUT settings
AcDbLayout
1      <<< layout name
Layout1
70      <<< flags bit-coded
1      <<< 1 = Indicates the PSLTSCALE value for this layout when this
...
layout is current
71      <<< Tab order ("Model" tab always appears as the first tab
...
regardless of its tab order)
1
10     <<< minimum limits for this layout (defined by LIMMIN while this
...
layout is current)
-0.25   <<< x value, distance of the left paper margin from the plot
...
origin-x, in plot paper units and by scale (e.g. x50 for 1:50)
20     <<< group code for y value

```

(continues on next page)

(continued from previous page)

```

-0.25    <<<      y value, distance of the bottom paper margin from the plot
...
11       <<< maximum limits for this layout (defined by LIMMAX while this
...
layout is current)
10.75    <<<      x value, distance of the right paper margin from the plot
...
21       <<<      group code for y value
8.25     <<<      y value, distance of the top paper margin from the plot
...
12       <<< insertion base point for this layout (defined by INSBASE while
...
this layout is current)
0.0      <<<      x value
22       <<<      group code for y value
0.0      <<<      y value
32       <<<      group code for z value
0.0      <<<      z value
14       <<< minimum extents for this layout (defined by EXTMIN while this
...
layout is current), AutoCAD default is (1e20, 1e20, 1e20)
1.05    <<<      x value
24       <<<      group code for y value
0.80    <<<      y value
34       <<<      group code for z value
0.0      <<<      z value
15       <<< maximum extents for this layout (defined by EXTMAX while this
...
layout is current), AutoCAD default is (-1e20, -1e20, -1e20)
9.45    <<<      x value
25       <<<      group code for y value
7.20    <<<      y value
35       <<<      group code for z value
0.0      <<<      z value
146     <<< elevation ???
0.0
13       <<< UCS origin (3D Point)
0.0      <<<      x value
23       <<<      group code for y value
0.0      <<<      y value
33       <<<      group code for z value
0.0      <<<      z value
16       <<< UCS X-axis (3D vector)
1.0      <<<      x value
26       <<<      group code for y value
0.0      <<<      y value
36       <<<      group code for z value
0.0      <<<      z value
17       <<< UCS Y-axis (3D vector)
0.0      <<<      x value
27       <<<      group code for y value
1.0      <<<      y value
37       <<<      group code for z value
0.0      <<<      z value
76       <<< orthographic type of UCS 0-6 (... too many options)
0        <<< 0 = UCS is not orthographic ???
330     <<< ID/handle of required block table record
58
331     <<< ID/handle to the viewport that was last active in this layout
...
when the layout was current
1B9

```

(continues on next page)

(continued from previous page)

```
1001    <<< extended data (ignore)
...

```

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

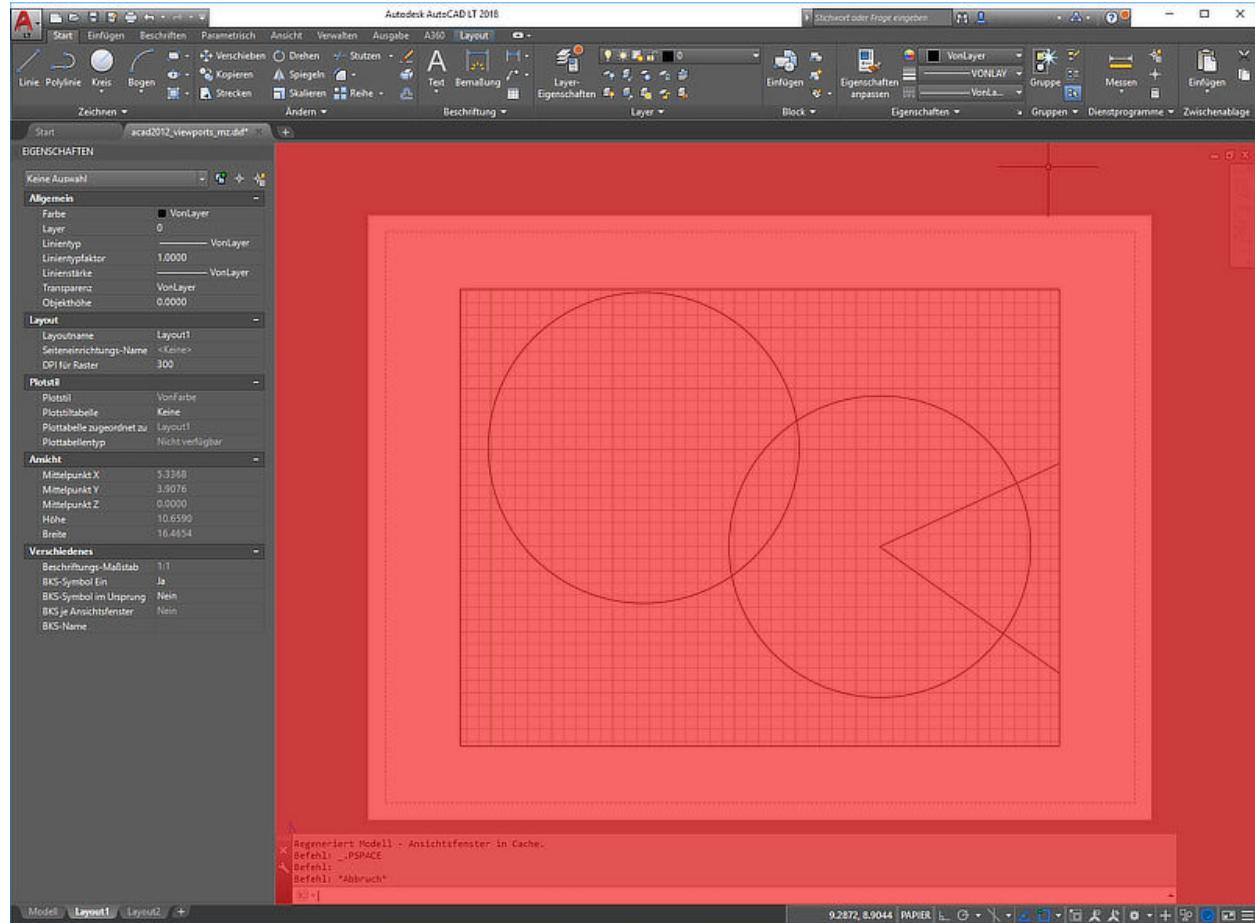
The modelspace layout requires (?) a VPORT entity in the VPORT table (group code 331 in the AcDbLayout subclass).

Main VIEWPORT Entity for LAYOUT

The “main” viewport for layout “Layout1” shown above. This viewport is located in the associated BLOCK definition called “*Paper_Space0”. Group code 330 in subclass AcDbLayout points to the BLOCK_RECORD of “*Paper_Space0”.

Remember: the entities of the *active* paperspace layout are located in the ENTITIES section, therefore “Layout1” is not the active paperspace layout.

The “main” VIEWPORT describes, how the application shows the paperspace layout on the screen, and I guess only AutoCAD needs this values.



```
0
VIEWPORT
5      <<< handle

```

(continues on next page)

(continued from previous page)

```

1B4
102      <<< extension dictionary (ignore)
{ACAD_XDICTIONARY
360
1B5
102
}
330      <<< owner handle
58       <<< points to BLOCK_RECORD (same as group code 330 in AcDbLayout of
...     "Layout1")
100
AcDbEntity
67       <<< paperspace flag
1       <<< 0 = modelspace; 1 = paperspace
8       <<< layer,
0
100
AcDbViewport
10      <<< Center point (in WCS)
5.25    <<<     x value
20      <<<     group code for y value
4.00    <<<     y value
30      <<<     group code for z value
0.0    <<<     z value
40      <<< width in paperspace units
23.55   <<< VIEW size in AutoCAD, depends on the workstation configuration
41      <<< height in paperspace units
9.00    <<< VIEW size in AutoCAD, depends on the workstation configuration
68      <<< viewport status field -1/0/n
2       <<< >0 On and active. The value indicates the order of stacking for
...     the viewports, where 1 is the active viewport, 2 is the next, and so forth
69      <<< viewport ID
1       <<< "main" viewport has always ID=1
12      <<< view center point in Drawing Coordinate System (DCS), defines
...     the center point of the VIEW in relation to the LAYOUT origin
5.25    <<<     x value
22      <<<     group code for y value
4.00    <<<     y value
13      <<< snap base point in modelspace
0.0    <<<     x value
23      <<<     group code for y value
0.0    <<<     y value
14      <<< snap spacing in modelspace units
0.5    <<<     x value
24      <<<     group code for y value
0.5    <<<     y value
15      <<< grid spacing in modelspace units
0.5    <<<     x value
25      <<<     group code for y value
0.5    <<<     y value
16      <<< view direction vector from target (in WCS)
0.0    <<<     x value
26      <<<     group code for y value
0.0    <<<     y value
36      <<<     group code for z value
1.0    <<<     z value
17      <<< view target point

```

(continues on next page)

(continued from previous page)

```

0.0      <<<    x value
27       <<<    group code for y value
0.0      <<<    y value
37       <<<    group code for z value
0.0      <<<    z value
42       <<<    perspective lens length, focal length?
50.0     <<<    50mm
43       <<<    front clip plane z value
0.0      <<<    z value
44       <<<    back clip plane z value
0.0      <<<    z value
45       <<<    view height (in modelspace units)
9.00
50       <<<    snap angle
0.0
51       <<<    view twist angle
0.0
72       <<<    circle zoom percent
1000
90       <<<    Viewport status bit-coded flags (... too many options)
819232  <<<    b11001000000000100000
1        <<<    plot style sheet name assigned to this viewport

281      <<<    render mode (... too many options)
0        <<<    0 = 2D optimized (classic 2D)
71       <<<    UCS per viewport flag
1        <<<    1 = This viewport stores its own UCS which will become the
...      current UCS whenever the viewport is activated
74       <<<    Display UCS icon at UCS origin flag
0        <<<    this field is currently being ignored and the icon always
...      represents the viewport UCS
110      <<<    UCS origin (3D point)
0.0      <<<    x value
120      <<<    group code for y value
0.0      <<<    y value
130      <<<    group code for z value
0.0      <<<    z value
111      <<<    UCS X-axis (3D vector)
1.0      <<<    x value
121      <<<    group code for y value
0.0      <<<    y value
131      <<<    group code for z value
0.0      <<<    z value
112      <<<    UCS Y-axis (3D vector)
0.0      <<<    x value
122      <<<    group code for y value
1.0      <<<    y value
132      <<<    group code for z value
0.0      <<<    z value
79       <<<    Orthographic type of UCS (... too many options)
0        <<<    0 = UCS is not orthographic
146      <<<    elevation
0.0
170      <<<    shade plot mode (0/1/2/3 for as displayed/wireframe/hidden/rendered)
0        <<<    as displayed
61       <<<    frequency of major grid lines compared to minor grid lines
5        <<<    major grid subdivided by 5

```

(continues on next page)

(continued from previous page)

```
348      <<< visual style ID/handle (optional)
9F
292      <<< default lighting flag, on when no user lights are specified.
1
282      <<< Default lighting type (0/1 = one distant light/two distant lights)
1      <<< one distant light
141      <<< view brightness
0.0
142      <<< view contrast
0.0
63       <<< ambient light color (ACI), write only if not black color
250
421      <<< ambient light color (RGB), write only if not black color
3355443
```

6.13 Developer Guides

Information about *ezdxf* internals.

6.13.1 Source Code Formatting

Reformat code by [Black](#) for a column width of 80:

```
C:\> black -l 80 <python-file>
```

Reformatting the *ezdxf* code base is an ongoing process, add reformatted code in a separate commit without changing the runtime logic.

6.13.2 Type Annotations

The use of type annotations is encouraged. New modules should pass [mypy](#) without errors in non-strict mode. Using `# type: ignore` is fine in tricky situations - type annotations should be helpful in understanding the code and not be a burden.

The following global options are required to pass [mypy](#) without error messages:

```
[mypy]
python_version = 3.7
ignore_missing_imports = True
```

Read [this](#) to learn where [mypy](#) searches for config files.

Use the [mypy](#) command line option `--ignore-missing-imports` and `-p` to check the whole package from any location in the file system:

```
PS D:\Source\ezdxf.git> mypy --ignore-missing-imports -p ezdxf
Success: no issues found in 255 source files
```

6.13.3 Design

The [Package Design for Developers](#) section shows the structure of the `ezdxf` package for developers with more experience, which want to have more insight into the package and maybe want to develop add-ons or want contribute to the `ezdxf` package.

!!! UNDER CONSTRUCTION !!!

Package Design for Developers

A DXF document is divided into several sections, these sections are managed by the `Drawing` object. For each section exist a corresponding attribute in the `Drawing` object:

Section	Attribute
HEADER	<code>Drawing.header</code>
CLASSES	<code>Drawing.classes</code>
TABLES	<code>Drawing.tables</code>
BLOCKS	<code>Drawing.blocks</code>
ENTITIES	<code>Drawing.entities</code>
OBJECTS	<code>Drawing.objects</code>

Resource entities (LAYER, STYLE, LTYPE, ...) are stored in tables in the TABLES section. A table owns the table entries, the owner handle of table entry is the handle of the table. Each table has a shortcut in the `Drawing` object:

Table	Attribute
APPID	<code>Drawing.appids</code>
BLOCK_RECORD	<code>Drawing.block_records</code>
DIMSTYLE	<code>Drawing.dimstyles</code>
LAYER	<code>Drawing.layers</code>
LTYPE	<code>Drawing.linetypes</code>
STYLE	<code>Drawing.styles</code>
UCS	<code>Drawing.ucs</code>
VIEW	<code>Drawing.views</code>
VPORT	<code>Drawing.viewports</code>

Graphical entities are stored in layouts: [Modelspace](#), [Paperspace](#) layouts and [BlockLayout](#). The core management object of these layouts is the `BLOCK_RECORD` entity (`BlockRecord`), the `BLOCK_RECORD` is the real owner of the entities, the owner handle of the entities is the handle of the `BLOCK_RECORD` and the `BLOCK_RECORD` also owns and manages the entity space of the layout which contains all entities of the layout.

For more information about layouts see also: [Layout Management Structures](#)

For more information about blocks see also: [Block Management Structures](#)

Non-graphical entities (objects) are stored in the OBJECTS section. Every object has a parent object in the OBJECTS section, most likely a DICTIONARY object, and is stored in the entity space of the OBJECTS section.

For more information about the OBJECTS section see also: [OBJECTS Section](#)

All table entries, DXF entities and DXF objects are stored in the entities database accessible as `Drawing.entitydb`. The entity database is a simple key, value storage, key is the entity handle, value is the DXF object.

For more information about the DXF data model see also: [Data Model](#)

Terminology

States

DXF entities and objects can have different states:

UNBOUND Entity is not stored in the Drawing entity database and DXF attribute handle is None and attribute doc can be None

BOUND Entity is stored in the Drawing entity database, attribute doc has a reference to Drawing and DXF attribute handle is not None

UNLINKED Entity is not linked to a layout/owner, DXF attribute owner is None

LINKED Entity is linked to a layout/owner, DXF attribute owner is not None

Virtual Entity State: UNBOUND & UNLINKED

Unlinked Entity State: BOUND & UNLINKED

Bound Entity State: BOUND & LINKED

Actions

NEW Create a new DXF document

LOAD Load a DXF document from an external source

CREATE Create DXF structures from NEW or LOAD data

DESTROY Delete DXF structures

BIND Bind an entity to a Drawing, set entity state to BOUND & UNLINKED and check or create required resources

UNBIND unbind ...

LINK Link an entity to an owner/layout. This makes an entity to a real DXF entity, which will be exported at the saving process. Any DXF entity can only be linked to **one** parent entity like DICTIONARY or BLOCK_RECORD.

UNLINK unlink ...

Loading a DXF Document

Loading a DXF document from an external source, creates a new Drawing object. This loading process has two stages:

First Loading Stage

- LOAD content from external source as SectionDict: `loader.load_dxf_structure()`
- LOAD tag structures as DXFEntity objects: `loader.load_dxf_entities()`
- BIND entities: `loader.load_and_bind_dxf_content()`; Special handling of the BIND process, because the Drawing is not full initialized, a complete validation is not possible at this stage.

Second Loading Stage

Parse SectionDict:

- CREATE sections: HEADER, CLASSES, TABLES, BLOCKS and OBJECTS
- CREATE layouts: Blocks, Layouts
- LINK entities to a owner/layout

The ENTITIES section is a relict from older DXF versions and has to be exported including the modelspace and active paperspace entities, but all entities reside in a BLOCK definition, even modelspace and paperspace layouts are only BLOCK definitions and ezdxf has no explicit ENTITIES section.

Source Code: as developer start your journey at `ezdxf.document.Drawing.read()`, which has no public documentation, because package-user should use `ezdxf.read()` and `ezdxf.readfile()`.

New DXF Document

Creating New DXF Entities

The default constructor of each entity type creates a new virtual entity:

- DXF attribute *owner* is None
- DXF attribute *handle* is None
- Attribute *doc* is None

The `DXFEntity.new()` constructor creates entities with given *owner*, *handle* and *doc* attributes, if *doc* is not None and entity is not already bound to a document, the `new()` constructor automatically bind the entity to the given document *doc*.

There exist only two scenarios:

1. UNBOUND: *doc* is None and *handle* is None
2. BOUND: *doc* is not None and *handle* is not None

Factory functions

- `new()`, create a new virtual DXF object/entity
- `load()`, load (create) virtual DXF object/entity from DXF tags
- `bind()`, bind an entity to a document, create required resources if necessary (e.g. ImageDefReactor, SEQEND) and raise exceptions for non-existing resources.
 - Bind entity loaded from an external source to a document, all referenced resources must exist, but try to repair as many flaws as possible because errors were created by another application and are not the responsibility of the package-user.
 - Bind an entity from another DXF document, all invalid resources will be removed silently or created (e.g. SEQEND). This is a simple import from another document without resource import, for a more advanced import including resources exist the `importer` add-on.
 - Bootstrap problem for binding loaded table entries and objects in the OBJECTS section! Can't use `Auditor` to repair this objects, because the DXF document is not fully initialized.
- `is_bound()` returns True if *entity* is bound to document *doc*

- `unbind()` function to remove an entity from a document and set state to a virtual entity, which should also *UNLINK* the entity from layout, because an layout can not store a virtual entity.
- `cls()`, returns the class
- `register_entity()`, registration decorator
- `replace_entity()`, registration decorator

Class Interfaces

DXF Entities

- NEW constructor to create an entity from scratch
- LOAD constructor to create an entity loaded from an external source
- DESTROY interface to kill an entity, set entity state to *dead*, which means `entity.is_alive` returns False. All entity iterators like `EntitySpace`, `EntityQuery`, and `EntityDB` must filter (ignore) *dead* entities. Calling `DXFEntity.destroy()` is a regular way to delete entities.
- LINK an entity to a layout by `BlockRecord.link()`, which set the *owner* handle to `BLOCK_RECORD` handle (= layout key) and add the entity to the entity space of the `BLOCK_RECORD` and set/clear the `perspace` flag.

DXF Objects

- NEW, LOAD, DESTROY see DXF entities
- LINK: Linking an DXF object means adding the entity to a parent object in the OBJECTS section, most likely a DICTIONARY object, and adding the object to the entity space of the OBJECTS section, the root-dict is the only entity in the OBJECTS section which has an invalid owner handle “0”. Any other object with an invalid or destroyed owner is an orphaned entity. The audit process destroys and removes orphaned objects.
- Extension dictionaries (ACAD_XDICTIONARY) are DICTIONARY objects located in the OBJECTS sections and can reference/own other entities of the OBJECTS section.
- The root-dictionary is the only entity in the OBJECTS section which has an invalid owner handle “0”. Any other object with an invalid or destroyed owner is an orphaned entity.

Layouts

- LINK interface to link an entity to a layout
- UNLINK interface to remove an entity from a layout

Database

- BIND interface to add an entity to the database of a document
- `delete_entity()` interface, same as UNBIND and DESTROY an entity

6.13.4 Internal Data Structures

Entity Database

The `EntityDB` is a simple key/value database to store `DXFEntity` objects by its handle, every `Drawing` has its own `EntityDB`, stored in the Drawing attribute `entitydb`.

Every DXF entity/object, except tables and sections, are represented as `DXFEntity` or inherited types, this entities are stored in the `EntityDB`, database-key is the `dxf.handle` as plain hex string.

All iterators like `keys()`, `values()`, `items()` and `__iter__()` do not yield destroyed entities.

Warning: The `get()` method and the index operator `[]`, return destroyed entities and entities from the trashcan.

```
class ezdxf.entitydb.EntityDB
```

`__getitem__(handle: str) → DXFEntity`
 Get entity by `handle`, does not filter destroyed entities nor entities in the trashcan.

`__setitem__(handle: str, entity: DXFEntity) → None`
 Set `entity` for `handle`.

`__delitem__(handle: str) → None`
 Delete entity by `handle`. Removes entity only from database, does not destroy the entity.

`__contains__(item: Union[str, DXFEntity]) → bool`
 True if database contains `handle`.

`__len__() → int`
 Count of database items.

`__iter__() → Iterator[str]`
 Iterable of all handles, does filter destroyed entities but not entities in the trashcan.

`get(handle: str) → Optional[DXFEntity]`
 Returns entity for `handle` or `None` if no entry exist, does not filter destroyed entities.

`next_handle() → str`
 Returns next unique handle.

`keys() → Iterable[str]`
 Iterable of all handles, does filter destroyed entities.

`values() → Iterable[DXFEntity]`
 Iterable of all entities, does filter destroyed entities.

`items() → Iterable[Tuple[str, DXFEntity]]`
 Iterable of all (handle, entities) pairs, does filter destroyed entities.

`add(entity: DXFEntity) → None`
 Add `entity` to database, assigns a new handle to the `entity` if `entity.dxf.handle` is `None`. Adding the same entity multiple times is possible and creates only a single database entry.

`new_trashcan() → ezdxf.entitydb.EntityDB.Trashcan`
 Returns a new trashcan, empty trashcan manually by: : func:`Trashcan.clear()`.

`trashcan() → ezdxf.entitydb.EntityDB.Trashcan`
 Returns a new trashcan in context manager mode, trashcan will be emptied when leaving context.

purge() → None
Remove all destroyed entities from database, but does not empty the trashcan.

Entity Space

class `ezdxf.entitydb.EntitySpace(entities: Iterable[ezdxf.entities.dxfentity.DXFEntity] = None)`
An `EntitySpace` is a collection of `DXFEntity` objects, that stores only references to `DXFEntity` objects.
The `Modelspace`, any `Paperspace` layout and `BlockLayout` objects have an `EntitySpace` container to store their entities.

__iter__() → Iterable[DXFEntity]
Iterable of all entities, filters destroyed entities.

__getitem__(index) → DXFEntity
Get entity at index `item`
`EntitySpace` has a standard Python list like interface, therefore `index` can be any valid list indexing or slicing term, like a single index `layout[-1]` to get the last entity, or an index slice `layout[:10]` to get the first 10 or less entities as `List[DXFEntity]`. Does not filter destroyed entities.

__len__() → int
Count of entities including destroyed entities.

has_handle(handle: str) → bool
True if `handle` is present, does filter destroyed entities.

purge()
Remove all destroyed entities from entity space.

add(entity: DXFEntity) → None
Add `entity`.

extend(entities: Iterable[DXFEntity]) → None
Add multiple `entities`.

remove(entity: DXFEntity) → None
Remove `entity`.

clear() → None
Remove all entities.

DXF Types

Required DXF tag interface:

- property `code`: group code as int
- property `value`: tag value of unspecific type
- `dxfstr()`: returns the DXF string
- `clone()`: returns a deep copy of tag

DXFTag Factory Functions

`ezdxf.lldxf.types.dxftag(code: int, value: Any) → ezdxf.lldxf.types.DXFTag`
DXF tag factory function.

Parameters

- **code** – group code
- **value** – tag value

Returns: *DXFTag* or inherited

```
ezdxf.lldxf.types.tuples_to_tags (iterable: Iterable[Tuple[int, Any]]) → Iterable[ezdxf.lldxf.types.DXFTag]
```

Returns an iterable if *DXFTag* or inherited, accepts an iterable of (code, value) tuples as input.

DXFTag

```
class eздxf.lldxf.types.DXFTag (code: int, value: Any)
```

Immutable DXFTag class.

Parameters

- **code** – group code as int
- **value** – tag value, type depends on group code

code

group code as int (do not change)

value

tag value (read-only property)

__eq__(other) → bool

True if *other* and *self* has same content for *code* and *value*.

__getitem__(index: int)

Returns *code* for index 0 and *value* for index 1, emulates a tuple.

__hash__()

Hash support, *DXFTag* can be used in sets and as dict key.

__iter__()

Returns (code, value) tuples.

__repr__() → str

Returns representation string 'DXFTag (code, value)'.

__str__() → str

Returns content string '(code, value)'.

clone() → eздxf.lldxf.types.DXFTag

Returns a clone of itself, this method is necessary for the more complex (and not immutable) DXF tag types.

dxftstr() → str

Returns the DXF string e.g. ' 0\nLINE\n'

DXFBinaryTag

```
class eздxf.lldxf.types.DXFBinaryTag (DXFTag)
```

Immutable BinaryTags class - immutable by design, not by implementation.

dxftstr() → str

Returns the DXF string for all vertex components.

tostring() → str

Returns binary value as single hex-string.

DXFVertex

class `ezdxf.lldxf.types.DXFVertex(DXFTag)`

Represents a 2D or 3D vertex, stores only the group code of the x-component of the vertex, because the y-group-code is x-group-code + 10 and z-group-code id x-group-code+20, this is a rule that ALWAYS applies. This tag is *immutable* by design, not by implementation.

Parameters

- **code** – group code of x-component
- **value** – sequence of x, y and optional z values

dxfrstr() → str

Returns the DXF string for all vertex components.

dxftags() → Iterable[ezdxf.lldxf.types.DXFTag]

Returns all vertex components as single `DXFTag` objects.

NONE_TAG

`ezdxf.lldxf.types.NONE_TAG`

Special tag representing a none existing tag.

Tags

A list of `DXFTag`, inherits from Python standard list. Unlike the statement in the DXF Reference “Do not write programs that rely on the order given here”, tag order is sometimes essential and some group codes may appear multiples times in one entity. At the worst case (Material: normal map shares group codes with diffuse map) using same group codes with different meanings.

class `ezdxf.lldxf.tags.Tags`

Subclass of `list`.

Collection of `DXFTag` as flat list. Low level tag container, only required for advanced stuff.

classmethod `from_text(text: str) → Tags`

Constructor from DXF string.

dxftype() → str

Returns DXF type of entity, e.g. 'LINE'.

get_handle() → str

Get DXF handle. Raises `DXFValueError` if handle not exist.

Returns handle as plain hex string like 'FF00'

Raises `DXFValueError` – no handle found

replace_handle(new_handle: str) → None

Replace existing handle.

Parameters `new_handle` – new handle as plain hex string e.g. 'FF00'

has_tag(code: int) → bool

Returns True if a `DXFTag` with given group `code` is present.

Parameters `code` – group code as int

has_embedded_objects() → bool

get_first_tag (*code: int, default=DXFValueError*) → DXFTag

Returns first *DXFTag* with given group code or *default*, if *default* != DXFValueError, else raises DXFValueError.

Parameters

- **code** – group code as int
- **default** – return value for default case or raises DXFValueError

get_first_value (*code: int, default=DXFValueError*) → Any

Returns value of first *DXFTag* with given group code or default if *default* != DXFValueError, else raises DXFValueError.

Parameters

- **code** – group code as int
- **default** – return value for default case or raises DXFValueError

find_all (*code: int*) → List[DXFTag]

Returns a list of *DXFTag* with given group code.

Parameters **code** – group code as int

filter (*codes: Iterable[int]*) → Iterable[DXFTag]

Iterate and filter tags by group *codes*.

Parameters **codes** – group codes to filter

collect_consecutive_tags (*codes: Iterable[int], start: int = 0, end: int = None*) → Tags

Collect all consecutive tags with group code in *codes*, *start* and *end* delimits the search range. A tag code not in *codes* ends the process.

Parameters

- **codes** – iterable of group codes
- **start** – start index as int
- **end** – end index as int, None for end index = len (self)

Returns collected tags as *Tags*

tag_index (*code: int, start: int = 0, end: int = None*) → int

Return index of first *DXFTag* with given group code.

Parameters

- **code** – group code as int
- **start** – start index as int
- **end** – end index as int, None for end index = len (self)

update (*tag: DXFTag*)

Update first existing tag with same group code as *tag*, raises DXFValueError if tag not exist.

set_first (*tag: DXFTag*)

Update first existing tag with group code *tag*.code or append tag.

remove_tags (*codes: Iterable[int]*) → None

Remove all tags inplace with group codes specified in *codes*.

Parameters **codes** – iterable of group codes as int

remove_tags_except (*codes: Iterable[int]*) → None
Remove all tags inplace except those with group codes specified in *codes*.

Parameters **codes** – iterable of group codes

pop_tags (*codes: Iterable[int]*) → Iterable[DXFTag]
Pop tags with group codes specified in *codes*.

Parameters **codes** – iterable of group codes

classmethod strip (*tags: Tags, codes: Iterable[int]*) → Tags
Constructor from *tags*, strips all tags with group codes in *codes* from tags.

Parameters

- **tags** – iterable of *DXFTag*
- **codes** – iterable of group codes as int

ezdxf.lldxf.tags.group_tags (*tags: Iterable[DXFTag], splitcode: int = 0*) → Iterable[Tags]
Group of tags starts with a SplitTag and ends before the next SplitTag. A SplitTag is a tag with code == *splitcode*, like (0, ‘SECTION’) for *splitcode* == 0.

Parameters

- **tags** – iterable of DXFTag
- **splitcode** – group code of split tag

class **ezdxf.lldxf.extendedtags.ExtendedTags** (*tags: Iterable[DXFTag]=None, legacy=False*)

Represents the extended DXF tag structure introduced with DXF R13.

Args: *tags*: iterable of *DXFTag* *legacy*: flag for DXF R12 tags

appdata

Application defined data as list of Tags

subclasses

Subclasses as list of Tags

xdata

XDATA as list of Tags

embedded_objects

embedded objects as list of Tags

noclass

Short cut to access first subclass.

get_handle() → str

Returns handle as hex string.

dxftype() → str

Returns DXF type as string like “LINE”.

replace_handle(handle: str) → None

Replace the existing entity handle by a new value.

legacy_repair()

Legacy (DXF R12) tags handling and repair.

clone() → ExtendedTags

Shallow copy.

flatten_subclasses()

Flatten subclasses in legacy mode (DXF R12).

There exists DXF R12 with subclass markers, technical incorrect but works if the reader ignore subclass marker tags, unfortunately ezdxf tries to use this subclass markers and therefore R12 parsing by ezdxf does not work without removing these subclass markers.

This method removes all subclass markers and flattens all subclasses into ExtendedTags.noclass.

get_subclass(name: str, pos: int = 0) → Tags

Get subclass *name*.

Parameters

- **name** – subclass name as string like “AcDbEntity”
- **pos** – start searching at subclass *pos*.

has_xdata(appid: str) → bool

True if has XDATA for *appid*.

get_xdata(appid: str) → Tags

Returns XDATA for *appid* as Tags.

set_xdata(appid: str, tags: IterableTags) → None

Set *tags* as XDATA for *appid*.

new_xdata(appid: str, tags: 'IterableTags' = None) → Tags

Append a new XDATA block.

Assumes that no XDATA block with the same *appid* already exist:

```
try:
    xdata = tags.get_xdata('EZDXF')
except ValueError:
    xdata = tags.new_xdata('EZDXF')
```

has_app_data(appid: str) → bool

True if has application defined data for *appid*.

get_app_data(appid: str) → Tags

Returns application defined data for *appid* as Tags including marker tags.

get_app_data_content(appid: str) → Tags

Returns application defined data for *appid* as Tags without first and last marker tag.

set_app_data_content(appid: str, tags: IterableTags) → None

Set application defined data for *appid* for already exiting data.

new_app_data(appid: str, tags: 'IterableTags' = None, subclass_name: str = None) → Tags

Append a new application defined data to subclass *subclass_name*.

Assumes that no app data block with the same *appid* already exist:

```
try:
    app_data = tags.get_app_data('{ACAD_REACTORS}', tags)
except ValueError:
    app_data = tags.new_app_data('{ACAD_REACTORS}', tags)
```

classmethod from_text(text: str, legacy: bool = False) → ExtendedTags

Create *ExtendedTags* from DXF text.

Packed DXF Tags

Store DXF tags in compact data structures as list or array.array to reduce memory usage.

class ezdxf.lldxf.packedtags.TagList(*data: Iterable = None*)

Store data in a standard Python list.

Args: *data*: iterable of DXF tag values.

values

Data storage as list.

clone() → TagList

Returns a deep copy.

classmethod from_tags(tags: Tags, code: int) → TagList

Setup list from iterable tags.

Parameters

- **tags** – tag collection as *Tags*

- **code** – group code to collect

clear() → None

Delete all data values.

class ezdxf.lldxf.packedtags.TagArray(*data: Iterable = None*)

TagArray is a subclass of *TagList*, which store data in an array.array. Array type is defined by class variable **DTYPE**.

Args: *data*: iterable of DXF tag values.

DTYPE

array.array type as string

values

Data storage as array.array

set_values(values: Iterable[T_co]) → None

Replace data by *values*.

class ezdxf.lldxf.packedtags.VertexArray(*data: Iterable = None*)

Store vertices in an array.array('d'). Vertex size is defined by class variable **VERTEX_SIZE**.

Args: *data*: iterable of vertex values as linear list e.g. [x1, y1, x2, y2, x3, y3, ...].

VERTEX_SIZE

Size of vertex (2 or 3 axis).

__len__() → int

Count of vertices.

__getitem__(index: int)

Get vertex at *index*, extended slicing supported.

__setitem__(index: int, point: Sequence[float]) → None

Set vertex *point* at *index*, extended slicing not supported.

__delitem__(index: int) → None

Delete vertex at *index*, extended slicing supported.

__iter__() → Iterator[Sequence[float]]

Returns iterable of vertices.

__str__() → str
String representation.

insert (pos: int, point: Sequence[float])
Insert *point* in front of vertex at index *pos*.

Parameters

- **pos** – insert position
- **point** – point as tuple

append (point: Sequence[float]) → None
Append *point*.

extend (points: Iterable[Sequence[float]]) → None
Extend array by *points*.

set (points: Iterable[Sequence[float]]) → None
Replace all vertices by *points*.

clear() → None
Delete all vertices.

clone() → VertexArray
Returns a deep copy.

classmethod from_tags (tags: Iterable[DXFTag], code: int = 10) → VertexArray
Setup point array from iterable tags.

Parameters

- **tags** – iterable of *DXFVertex*
- **code** – group code to collect

export_dxf (tagwriter: *ezdxf.lldxf.tagwriter.TagWriter*, code=10)

XData

class *ezdxf.entities.xdata.XData*
Internal management class for XDATA.

See also:

- XDATA user reference: *Extended Data (XDATA)*
- Wrapper class to store a list in XDATA: *XDataUserList*
- Wrapper class to store a dict in XDATA: *XDataUserDict*
- Tutorial: *Storing Custom Data in DXF Files*
- DXF Internals: *Extended Data*
- **DXF R2018 Reference**

__contains__ (appid: str) → bool
Returns True if DXF tags for *appid* exist.

add (appid: str, tags: Iterable[Union[Tuple[int, Any], DXFTag]]) → None
Add a list of DXF tags for *appid*. The *tags* argument is an iterable of (group code, value) tuples, where the group code has to be an integer value. The mandatory XDATA marker (1001, appid) is added automatically if front of the tags if missing.

Each entity can contain only one list of tags for each *appid*. Adding a second list of tags for the same *appid* replaces the existing list of tags.

The valid XDATA group codes are restricted to some specific values in the range from 1000 to 1071, for more information see also the internals about [Extended Data](#).

get (*appid*: str) → eздxf.lldxfs.Tags

Returns the DXF tags as [Tags](#) list stored by *appid*.

Raises DXFValueError – no data for *appid* exist

discard (*appid*)

Delete DXF tags for *appid*. None existing appids are silently ignored.

has_xlist (*appid*: str, *name*: str) → bool

Returns True if list *name* from XDATA *appid* exists.

Parameters

- **appid** – APPID
- **name** – list name

get_xlist (*appid*: str, *name*: str) → List[Tuple]

Get list *name* from XDATA *appid*.

Parameters

- **appid** – APPID
- **name** – list name

Returns: list of DXFTags including list name and curly braces '{' '}' tags

Raises

- DXFKeyError – XDATA *appid* does not exist
- DXFValueError – list *name* does not exist

set_xlist (*appid*: str, *name*: str, *tags*: Iterable[T_co]) → None

Create new list *name* of XDATA *appid* with *xdata_tags* and replaces list *name* if already exists.

Parameters

- **appid** – APPID
- **name** – list name
- **tags** – list content as DXFTags or (code, value) tuples, list name and curly braces '{' '}' tags will be added

discard_xlist (*appid*: str, *name*: str) → None

Deletes list *name* from XDATA *appid*. Ignores silently if XDATA *appid* or list *name* not exist.

Parameters

- **appid** – APPID
- **name** – list name

replace_xlist (*appid*: str, *name*: str, *tags*: Iterable[T_co]) → None

Replaces list *name* of existing XDATA *appid* by *tags*. Appends new list if list *name* do not exist, but raises DXFValueError if XDATA *appid* do not exist.

Low level interface, if not sure use *set_xdata_list()* instead.

Parameters

- **appid** – APPID
- **name** – list name
- **tags** – list content as DXFTags or (code, value) tuples, list name and curly braces ‘{’ ‘}’ tags will be added

Raises DXFValueError – XDATA *appid* do not exist

transform(*m*: Matrix44) → None

Transform XDATA tags with group codes 1011, 1012, 1013, 1041 and 1042 inplace. For more information see [Extended Data](#) Internals.

Application-Defined Data (AppData)

Starting at DXF R13, DXF objects can contain application-defined codes (AppData) outside of XDATA.

All AppData is defined with a beginning (102, “{APPID”) tag and according to the DXF reference appear should appear before the first subclass marker.

There are two known use cases of this data structure in Autodesk products:

- ACAD_REACTORS, store handles to persistent reactors in a DXF entity
- ACAD_XDICTIONARY, store handle to the extension dictionary of a DXF entity

Both AppIDs are not defined/stored in the AppID table!

class ezdxf.entities.appdata.**AppData**

Internal management class for Application defined data.

See also:

- User reference: [Application-Defined Data \(AppData\)](#)
- Internals about [Application-Defined Codes](#) tags

__contains__(*appid*: str) → bool

Returns True if application-defined data exist for *appid*.

__len__() → int

Returns the count of AppData.

add(*appid*: str, *data*: Iterable[Sequence[T_co]]) → None

Add application-defined tags for *appid*. Adds first tag (102, “{APPID”) if not exist. Adds last tag (102, “}” if not exist.

get(*appid*: str) → ezdxf.llodxf.tags.Tags

Get application-defined data for *appid* as [Tags](#) container. The first tag is always (102, “{APPID”). The last tag is always (102, “}”).

set(*tags*: ezdxf.llodxf.tags.Tags) → None

Store raw application-defined data tags. The first tag has to be (102, “{APPID”). The last tag has to be (102, “}”).

discard(*appid*: str)

Delete application-defined data for *appid* without raising and error if *appid* doesn’t exist.

Reactors

```
class ezdxf.entities.appdata.Reactors
```

Internal management class for persistent reactor handles. Handles are stored as hex strings like "ABBA".

See also:

- User reference: [Reactors](#)
- Internals about [Persistent Reactors](#) tags

__contains__(handle: str) → bool
Returns True if *handle* is registered.

__len__() → int
Returns count of registered handles.

__iter__() → Iterator[str]
Returns an iterator for all registered handles.

add(handle: str) → None
Add a single *handle*.

get() → List[str]
Returns all registered handles as sorted list.

set(handles: Optional[Iterable[str]]) → None
Reset all handles.

discard(handle: str)
Discard a single *handle*.

6.13.5 Documentation Guide

Formatting Guide

This section is only for myself, because of the long pauses between develop iterations, I often forget to be consistent in documentation formatting.

Documentation is written with [Sphinx](#) and [reStructuredText](#).

Started integration of documentation into source code and using [autodoc](#) features of [Sphinx](#) wherever useful.

Sphinx theme provided by [Read the Docs](#) :

```
pip install sphinx-rtd-theme
```

guide — Example module

```
guide.example_func(a:int, b:str, test:str=None, flag:bool=True) → None
```

Parameters *a* and *b* are positional arguments, argument *test* defaults to None and *flag* to True. Set *a* to 70 and *b* to "x" as an example. Inline code examples `example_func(70, 'x')` or simple `example_func(70, "x")`

- arguments: *a*, *b*, *test* and *flags*
- literal number values: 1, 2 ... 999
- literal string values: "a String"

- literal tags: (5, “F000”)
- inline code: call a `example_func(x)`
- Python keywords: None, True, False, tuple, list, dict, str, int, float
- Exception classes: DXFAttributeError

```
class guide.ExampleCls(**kwargs)
```

The `ExampleCls` constructor accepts a number of optional keyword arguments. Each keyword argument corresponds to an instance attribute, so for example

```
e = ExampleCls(flag=True)
```

flag

This is the attribute `flag`.

set_axis(axis)

axis as (x, y, z) tuple

Args: axis: (x, y, z) tuple

example_method(flag:bool=False) → None

Method `example_method()` of class `ExampleCls`

Text Formatting

DXF version DXF R12 (AC1009), DXF R2004 (AC1018)

DXF Types DXF types are always written in uppercase letters but without further formatting: DXF, LINE, CIRCLE

(internal API) Marks methods as internal API, gets no public documentation.

(internal class) Marks classes only for internal usage, gets no public documentation.

Spatial Dimensions 2D and 3D with an uppercase letter D

Axis x-axis, y-axis and z-axis

Planes xy-plane, xz-plane, yz-plane

Layouts modelspace, paperspace [layout], block [layout]

Extended Entity Data AppData, XDATA, embedded object, APPID

6.14 Glossary

ACI AutoCAD Color Index (ACI)

ACIS The 3D ACIS Modeler (ACIS) is a geometric modeling kernel developed by Spatial Corp. ® (formerly Spatial Technology) and now part of Dassault Systems. All ACIS based DXF entities store their geometry as SAT or SAB data. These are not open data formats and a license has to be purchased to get access to their SDK, therefore ezdxf can not provide any support for creating, processing or transforming of ACIS based DXF entities.

bulge The *Bulge value* is used to create arc shaped line segments in `Polyline` and `LWPolyline` entities.

CAD Computer-Assisted Drafting or Computer-Aided Design

CTB Color dependent plot style table (ColorDependentPlotStyles)

DWG Proprietary file format of AutoCAD®. Documentation for this format is available from the Open Design Alliance (ODA) at their [Downloads](#) section. This documentation is created by reverse engineering therefore not perfect nor complete.

DXF Drawing eXchange Format is a file format used by AutoCAD® to interchange data with other CAD applications. DXF is a trademark of Autodesk®. See also [What is DXF?](#)

raw color Raw color value as stored in DWG files, this integer value can represent ACI values as well as and *true color* values

reliable CAD application CAD applications which create valid DXF documents in the meaning and interpretation of Autodesk. See also [What is DXF?](#)

SAB ACIS file format, data stored as binary data

SAT ACIS file format, data stored as ASCII text

STB Named plot style table (`NamedPlotStyles`)

true color RGB color representation, a combination red, green and blue values to define a color.

6.15 Indices and tables

- genindex
- search

Python Module Index

e

ezdxf.addons, 540
ezdxf.addons.acadctb, 574
ezdxf.addons.drawing, 540
ezdxf.addons.dxf2code, 553
ezdxf.addons.geo, 545
ezdxf.addons.importer, 549
ezdxf.addons.iterdx, 555
ezdxf.addons.odafc, 564
ezdxf.addons.pycsg, 568
ezdxf.addons.r12writer, 558
ezdxf.addons.text2path, 566
ezdxf.bbox, 476
ezdxf.blkrefs, 391
ezdxf.colors, 402
ezdxf.comments, 510
ezdxf.disassemble, 474
ezdxf.document, 199
ezdxf.entities, 280
ezdxf.entities.appdata, 697
ezdxf.entities.dxfgroups, 279
ezdxf.entities.xdata, 695
ezdxf.entities.xdict, 389
ezdxf.entitydb, 687
ezdxf.enums, 397
ezdxf.gfxattribs, 490
ezdxf.layouts, 249
ezdxf.llodxf.const, 392
ezdxf.llodxf.extendedtags, 692
ezdxf.llodxf.packedtags, 693
ezdxf.llodxf.tags, 690
ezdxf.llodxf.types, 688
ezdxf.math, 407
ezdxf.options, 503
ezdxf.path, 461
ezdxf.query, 404
ezdxf.recover, 204
ezdxf.render, 523
ezdxf.render.arrows, 540

ezdxf.render.forms, 527
ezdxf.render.point, 539
ezdxf.render.trace, 537
ezdxf.reorder, 510
ezdxf.sections.blocks, 212
ezdxf.sections.classes, 210
ezdxf.sections.entities, 213
ezdxf.sections.header, 209
ezdxf.sections.objects, 213
ezdxf.sections.table, 216
ezdxf.sections.tables, 211
ezdxf.tools.fonts, 500
ezdxf.tools.text, 492
ezdxf.tools.text_size, 499
ezdxf.units, 35
ezdxf.upright, 480
ezdxf.urecord, 486
ezdxf.zoom, 509

g

guide, 698

Symbols

—abs__(*ezdxf.math.Vec3 method*), 429
—add__(*ezdxf.addons.pycsg.CSG method*), 573
—add__(*ezdxf.math.Matrix method*), 459
—add__(*ezdxf.math.Vec3 method*), 430
—bool__(*ezdxf.math.Vec3 method*), 430
—contains__(*ezdxf.document.ezdx.dfx.document.MetaData method*), 198
—contains__(*ezdxf.entities.Dictionary method*), 374
—contains__(*ezdxf.entities.appdata.AppData method*), 697
—contains__(*ezdxf.entities.appdata.Reactors method*), 698
—contains__(*ezdxf.entities.dxfgroups.DXFGGroup method*), 279
—contains__(*ezdxf.entities.dxfgroups.GroupCollection method*), 280
—contains__(*ezdxf.entities.xdata.XData method*), 695
—contains__(*ezdxf.entities.xdict.ExtensionDict method*), 390
—contains__(*ezdxf.entitydb.EntityDB method*), 687
—contains__(*ezdxf.layouts.BlockLayout method*), 278
—contains__(*ezdxf.layouts.Layout method*), 275
—contains__(*ezdxf.layouts.Layouts method*), 249
—contains__(*ezdxf.sections.blocks.BlocksSection method*), 212
—contains__(*ezdxf.sections.header.HeaderSection method*), 209
—contains__(*ezdxf.sections.objects.ObjectsSection method*), 214
—contains__(*ezdxf.sections.table.Table method*), 216
—copy__(*ezdxf.math.Matrix44 method*), 425
—copy__(*ezdxf.math.Vec3 method*), 429
—deepcopy__(*ezdxf.math.Vec3 method*), 429
—delitem__(*ezdxf.addons.acadctb.NamedPlotStyles method*), 576
—delitem__(*ezdxf.document.ezdx.dfx.document.MetaData method*), 199
—delitem__(*ezdxf.entities.Dictionary method*), 374
—delitem__(*ezdxf.entities.MetaData method*), 295
—delitem__(*ezdxf.entities.LWPolyline method*), 320
—delitem__(*ezdxf.entities.xdata.XDataUserDict method*), 486
—delitem__(*ezdxf.entities.xdata.XDataUserList method*), 485
—delitem__(*ezdxf.entities.xdict.ExtensionDict method*), 390
—delitem__(*ezdxf.entitydb.EntityDB method*), 687
—delitem__(*ezdxf.lldx.dfx.packedtags.VertexArray method*), 694
—delitem__(*ezdxf.sections.blocks.BlocksSection method*), 212
—delitem__(*ezdxf.sections.header.HeaderSection method*), 209
—eq__(*ezdxf.lldx.dfx.types.DXFTag method*), 689
—eq__(*ezdxf.math.Matrix method*), 459
—eq__(*ezdxf.math.Vec3 method*), 430
—geo_interface__(*ezdxf.addons.geo.GeoProxy attribute*), 547
—getitem__(*ezdxf.addons.acadctb.ColorDependentPlotStyles method*), 575
—getitem__(*ezdxf.addons.acadctb.NamedPlotStyles method*), 576
—getitem__(*ezdxf.document.ezdx.dfx.document.MetaData method*), 199
—getitem__(*ezdxf.entities.Dictionary method*), 374
—getitem__(*ezdxf.entities.DimStyleOverride method*), 295
—getitem__(*ezdxf.entities.LWPolyline method*),

320
—getitem__() (ezdxf.entities.MeshVertexCache
method), 349
—getitem__() (ezdxf.entities.Polyline method), 346
—getitem__() (ezdxf.entities.dxfgroups.DXFGroup
method), 279
—getitem__() (ezdxf.entities.mline.ezdx.
entities.mline.MLineStyleElmer(ezdx.
MLineStyleUserDependent method), 487
method), 325
—getitem__() (ezdxf.entities.xdata.XDataUserDict
method), 486
—getitem__() (ezdxf.entities.xdata.XDataUserList
method), 484
—getitem__() (ezdxf.entities.xdict.ExtensionDict
method), 390
—getitem__() (ezdxf.entitydb.EntityDB method),
687
—getitem__() (ezdxf.entitydb.EntitySpace method),
688
—getitem__() (ezdxf.layouts.BaseLayout method),
251
—getitem__() (ezdxf.lldx.
packedtags.VertexArray
method), 694
—getitem__() (ezdxf.lldx.
types.DXFTag method),
689
—getitem__() (ezdxf.math.ConstructionBox
method), 445
—getitem__() (ezdxf.math.Matrix method), 459
—getitem__() (ezdxf.math.Matrix44 method), 427
—getitem__() (ezdxf.math.Shape2d method), 447
—getitem__() (ezdxf.math.Vec3 method), 429
—getitem__() (ezdxf.query.EntityQuery method),
405
—getitem__() (ezdxf.render.trace.TraceBuilder
method), 538
—getitem__() (ezdxf.sections.blocks.BlocksSection
method), 212
—getitem__() (ezdxf.sections.header.HeaderSection
method), 209
—getitem__() (ezdxf.sections.objects.ObjectsSection
method), 214
—hash__() (ezdxf.lldx.
types.DXFTag method), 689
—hash__() (ezdxf.math.Matrix44 method), 426
—hash__() (ezdxf.math.Vec3 method), 429
—iadd__() (ezdxf.entities.MText method), 333
—iadd__() (ezdxf.tools.text.MTextEditor method),
492
—imul__() (ezdxf.math.Matrix44 method), 427
—init__() (ezdxf.addons.drawing.ezdx.
addons.drawing.matplotlib.MatplotlibBackend
method), 541
—init__() (ezdxf.addons.drawing.ezdx.
addons.drawing.pyqt.QBackend
method), 541
—init__() (ezdxf.entities.xdata.XDataUserDict
method), 485
—init__() (ezdxf.entities.xdata.XDataUserList
method), 484
—init__() (ezdxf.render.EulerSpiral method), 526
—init__() (ezdxf.render.R12Spline method), 525
—init__() (ezdxf.render.Spline method), 523
—init__() (ezdxf.urecord.BinaryRecord method),
488
—iter__() (ezdxf.addons.acadctb.ColorDependentPlotStyles
method), 575
—iter__() (ezdxf.addons.acadctb.NamedPlotStyles
method), 576
—iter__() (ezdxf.addons.geo.GeoProxy method),
548
—iter__() (ezdxf.entities.LWPolyline method), 320
—iter__() (ezdxf.entities.appdata.Reactors method),
698
—iter__() (ezdxf.entities.dxfgroups.DXFGroup
method), 279
—iter__() (ezdxf.entities.dxfgroups.GroupCollection
method), 280
—iter__() (ezdxf.entities.xdata.XDataUserDict
method), 486
—iter__() (ezdxf.entitydb.EntityDB method), 687
—iter__() (ezdxf.entitydb.EntitySpace method), 688
—iter__() (ezdxf.gfxattribs.GfxAtribs method), 491
—iter__() (ezdxf.layouts.BaseLayout method), 251
—iter__() (ezdxf.layouts.Layouts method), 249
—iter__() (ezdxf.lldx.
packedtags.VertexArray
method), 694
—iter__() (ezdxf.lldx.
types.DXFTag method), 689
—iter__() (ezdxf.math.ConstructionBox method),
445
—iter__() (ezdxf.math.Matrix44 method), 427
—iter__() (ezdxf.math.Vec3 method), 429
—iter__() (ezdxf.query.EntityQuery method), 405
—iter__() (ezdxf.sections.blocks.BlocksSection
method), 212
—iter__() (ezdxf.sections.entities.EntitySection
method), 213
—iter__() (ezdxf.sections.header.CustomVars
method), 209
—iter__() (ezdxf.sections.objects.ObjectsSection
method), 214
—iter__() (ezdxf.sections.table.Table method), 216
—len__() (ezdxf.entities.Dictionary method), 374
—len__() (ezdxf.entities.LWPolyline method), 320
—len__() (ezdxf.entities.MLine method), 323
—len__() (ezdxf.entities.mline.MLine method), 346
—len__() (ezdxf.entities.appdata.AppData method),
488
—len__() (ezdxf.entities.appdata.Reactors method),
698
—len__() (ezdxf.entities.dxfgroups.DXFGroup
method), 279

`__len__()` (*ezdxf.entities.dxfgroups.GroupCollection method*), 280
`__len__()` (*ezdxf.entities.mline.ezdxfs.entities.mline.MLineStyleElements method*), 325
`__len__()` (*ezdxf.entities.xdata.XDataUserDict method*), 486
`__len__()` (*ezdxf.entities.xdata.XDataUserList method*), 484
`__len__()` (*ezdxf.entities.xdict.ExtensionDict method*), 390
`__len__()` (*ezdxf.entitydb.EntityDB method*), 687
`__len__()` (*ezdxf.entitydb.EntitySpace method*), 688
`__len__()` (*ezdxf.layouts.BaseLayout method*), 251
`__len__()` (*ezdxf.layouts.Layouts method*), 249
`__len__()` (*ezdxf.lldxfs.packedtags.VertexArray method*), 694
`__len__()` (*ezdxf.math.Shape2d method*), 447
`__len__()` (*ezdxf.math.Vec3 method*), 429
`__len__()` (*ezdxf.query.EntityQuery method*), 405
`__len__()` (*ezdxf.render.trace.TraceBuilder method*), 538
`__len__()` (*ezdxf.sections.entities.EntitySection method*), 213
`__len__()` (*ezdxf.sections.header.CustomVars method*), 209
`__len__()` (*ezdxf.sections.header.HeaderSection method*), 209
`__len__()` (*ezdxf.sections.objects.ObjectsSection method*), 214
`__len__()` (*ezdxf.sections.table.Table method*), 216
`__lt__()` (*ezdxf.math.Vec3 method*), 430
`__mul__()` (*ezdxf.addons.pycsg.CSG method*), 574
`__mul__()` (*ezdxf.math.Matrix method*), 459
`__mul__()` (*ezdxf.math.Matrix44 method*), 427
`__mul__()` (*ezdxf.math.Vec3 method*), 430
`__neg__()` (*ezdxf.math.Vec3 method*), 430
`__radd__()` (*ezdxf.math.Vec3 method*), 430
`__repr__()` (*ezdxf.entities.DXFEntity method*), 281
`__repr__()` (*ezdxf.gfxattribs.GfxAttribs method*), 491
`__repr__()` (*ezdxf.lldxfs.types.DXFTag method*), 689
`__repr__()` (*ezdxf.math.ConstructionBox method*), 445
`__repr__()` (*ezdxf.math.Matrix44 method*), 424
`__repr__()` (*ezdxf.math.Vec3 method*), 429
`__rmul__()` (*ezdxf.math.Vec3 method*), 430
`__rsub__()` (*ezdxf.math.Vec3 method*), 430
`__setitem__()` (*ezdxf.document.ezdxfs.document.MetaData method*), 199
`__setitem__()` (*ezdxf.entities.Dictionary method*), 374
`__setitem__()` (*ezdxf.entities.DimStyleOverride method*), 295
`__setitem__()` (*ezdxf.entities.LWPolyline method*), 320
`__setitem__()` (*ezdxf.entities.MeshVertexCache method*), 350
`__setitem__()` (*ezdxf.entities.MLineStyleElements method*), 486
`__setitem__()` (*ezdxf.entities.xdata.XDataUserList method*), 484
`__setitem__()` (*ezdxf.entities.xdict.ExtensionDict method*), 390
`__setitem__()` (*ezdxf.entitydb.EntityDB method*), 687
`__setitem__()` (*ezdxf.lldxfs.packedtags.VertexArray method*), 694
`__setitem__()` (*ezdxf.math.Matrix method*), 459
`__setitem__()` (*ezdxf.math.Matrix44 method*), 427
`__setitem__()` (*ezdxf.sections.header.HeaderSection method*), 209
`__str__()` (*ezdxf.entities.DXFEntity method*), 281
`__str__()` (*ezdxf.entities.xdata.XDataUserDict method*), 486
`__str__()` (*ezdxf.entities.xdata.XDataUserList method*), 484
`__str__()` (*ezdxf.gfxattribs.GfxAttribs method*), 491
`__str__()` (*ezdxf.lldxfs.packedtags.VertexArray method*), 694
`__str__()` (*ezdxf.lldxfs.types.DXFTag method*), 689
`__str__()` (*ezdxf.math.ConstructionCircle method*), 437
`__str__()` (*ezdxf.math.ConstructionLine method*), 436
`__str__()` (*ezdxf.math.ConstructionRay method*), 435
`__str__()` (*ezdxf.math.Vec3 method*), 429
`__str__()` (*ezdxf.tools.text.MTextEditor method*), 493
`__str__()` (*ezdxf.urecord.BinaryRecord method*), 488
`__str__()` (*ezdxf.urecord.UserRecord method*), 488
`__sub__()` (*ezdxf.addons.pycsg.CSG method*), 573
`__sub__()` (*ezdxf.math.Matrix method*), 459
`__sub__()` (*ezdxf.math.Vec3 method*), 430
`__truediv__()` (*ezdxf.math.Vec3 method*), 430

A

`abs_tol` (*ezdxf.render.trace.LinearTrace attribute*), 538
`abs_tol` (*ezdxf.render.trace.TraceBuilder attribute*), 537
`AbstractFont` (*class in ezdxf.tools.fonts*), 500
`acad_release` (*ezdxf.document.Drawing attribute*), 199
`ACCURATE` (*ezdxf.addons.drawing.config.LinePolicy attribute*), 543
`ACI`, 699
`ACI` (*class in ezdxf.enums*), 401
`aci` (*ezdxf.addons.acadctb.PlotStyle attribute*), 577
`aci()` (*ezdxf.tools.text.MTextEditor method*), 493
`aci2rgb()` (*in module ezdxf.colors*), 402
`ACIS`, 699
`acis_data` (*ezdxf.entities.Body attribute*), 290

active_layout() (*ezdxf.document.Drawing method*), 202
active_layout() (*ezdxf.layouts.Layouts method*), 250
actual_measurement (*ezdxf.entities.Dimension.dxf attribute*), 294
adaptive_linenode (*ezdxf.addons.acadctb.PlotStyle attribute*), 577
add() (*ezdxf.entities.appdata.AppData method*), 697
add() (*ezdxf.entities.appdata.Reactors method*), 698
add() (*ezdxf.entities.Dictionary method*), 375
add() (*ezdxf.entities.XData method*), 695
add() (*ezdxf.entitydb.EntityDB method*), 687
add() (*ezdxf.entitydb.EntitySpace method*), 688
add() (*ezdxf.sections.table.AppIDTable method*), 219
add() (*ezdxf.sections.table.BlockRecordTable method*), 220
add() (*ezdxf.sections.table.DimStyleTable method*), 219
add() (*ezdxf.sections.table.LayerTable method*), 217
add() (*ezdxf.sections.table.LinetypeTable method*), 217
add() (*ezdxf.sections.table.TextstyleTable method*), 218
add() (*ezdxf.sections.table.UCSTable method*), 219
add() (*ezdxf.sections.table.ViewportTable method*), 220
add() (*ezdxf.sections.table.ViewTable method*), 219
add_3dface() (*ezdxf.addons.r12writer.R12FastStreamWriter method*), 561
add_3dface() (*ezdxf.layouts.BaseLayout method*), 254
add_3dsolid() (*ezdxf.layouts.BaseLayout method*), 273
add_aligned_dim() (*ezdxf.layouts.BaseLayout method*), 263
add_angular_dim_21() (*ezdxf.layouts.BaseLayout method*), 266
add_angular_dim_3p() (*ezdxf.layouts.BaseLayout method*), 267
add_angular_dim_arc() (*ezdxf.layouts.BaseLayout method*), 268
add_angular_dim_cra() (*ezdxf.layouts.BaseLayout method*), 268
add_arc() (*ezdxf.addons.r12writer.R12FastStreamWriter method*), 561
add_arc() (*ezdxf.entities.EdgePath method*), 309
add_arc() (*ezdxf.layouts.BaseLayout method*), 253
add_arc_dim_3p() (*ezdxf.layouts.BaseLayout method*), 269
add_arc_dim_arc() (*ezdxf.layouts.BaseLayout method*), 270
add_arc_dim_cra() (*ezdxf.layouts.BaseLayout method*), 270
add_attdef() (*ezdxf.layouts.BaseLayout method*), 255
add_attrib() (*ezdxf.entities.Insert method*), 244
add_auto_attribs() (*ezdxf.entities.Insert method*), 245
add_auto_blockref() (*ezdxf.layouts.BaseLayout method*), 255
add_bezier3p() (*in module ezdxf.path*), 469
add_bezier4p() (*in module ezdxf.path*), 470
add_blockref() (*ezdxf.layouts.BaseLayout method*), 254
add_body() (*ezdxf.layouts.BaseLayout method*), 273
add_cad_spline_control_frame() (*ezdxf.layouts.BaseLayout method*), 259
add_circle() (*ezdxf.addons.r12writer.R12FastStreamWriter method*), 561
add_circle() (*ezdxf.layouts.BaseLayout method*), 253
add_class() (*ezdxf.sections.classes.ClassesSection method*), 210
add_diameter_dim() (*ezdxf.layouts.BaseLayout method*), 265
add_diameter_dim_2p() (*ezdxf.layouts.BaseLayout method*), 266
add_dict_var() (*ezdxf.entities.Dictionary method*), 375
add_dictionary() (*ezdxf.entities.xdict.ExtensionDict method*), 391
add_dictionary() (*ezdxf.sections.objects.ObjectsSection method*), 214
add_dictionary_var() (*ezdxf.entities.xdict.ExtensionDict method*), 391
add_dictionary_var() (*ezdxf.sections.objects.ObjectsSection method*), 214
add_dictionary_with_default() (*ezdxf.sections.objects.ObjectsSection method*), 214
add_edge_creature() (*ezdxf.entities.MeshData method*), 327
add_edge_path() (*ezdxf.entities.BoundaryPaths method*), 307
add_ellipse() (*ezdxf.entities.EdgePath method*), 309
add_ellipse() (*ezdxf.layouts.BaseLayout method*), 253
add_ellipse() (*in module ezdxf.path*), 470
add_entity() (*ezdxf.layouts.BaseLayout method*), 252
add_extruded_surface() (*ezdxf.layouts.BaseLayout method*), 273
add_face() (*ezdxf.entities.MeshData method*), 327
add_face() (*ezdxf.render.MeshBuilder method*), 533
add_foreign_entity() (*ezdxf.layouts.BaseLayout method*), 252
add_geodata() (*ezdxf.sections.objects.ObjectsSection*

method), 214
 add_hatch () (*ezdxf.layouts.BaseLayout method*), 260
 add_image () (*ezdxf.layouts.BaseLayout method*), 261
 add_image_def () (*ezdxf.document.Drawing method*), 202
 add_image_def () (*ezdxf.sections.objects.ObjectsSection method*), 215
 add_import () (*ezdxf.addons.dxf2code.Code method*), 555
 add_leader () (*ezdxf.layouts.BaseLayout method*), 272
 add_line () (*ezdxf.addons.dxf2code.Code method*), 555
 add_line () (*ezdxf.addons.r12writer.R12FastStreamWriter method*), 560
 add_line () (*ezdxf.entities.EdgePath method*), 309
 add_line () (*ezdxf.entities.Pattern method*), 312
 add_line () (*ezdxf.layouts.BaseLayout method*), 253
 add_linear_dim () (*ezdxf.layouts.BaseLayout method*), 261
 add_lines () (*ezdxf.addons.dxf2code.Code method*), 555
 add_lofted_surface () (*ezdxf.layouts.BaseLayout method*), 274
 add_lwpolyline () (*ezdxf.layouts.BaseLayout method*), 256
 add_mesh () (*ezdxf.layouts.BaseLayout method*), 261
 add_mesh () (*ezdxf.render.MeshBuilder method*), 533
 add_mline () (*ezdxf.layouts.BaseLayout method*), 258
 add_mpolygon () (*ezdxf.layouts.BaseLayout method*), 260
 add_mtext () (*ezdxf.layouts.BaseLayout method*), 257
 add_mtext_dynamic_auto_height_columns () (*ezdxf.layouts.BaseLayout method*), 258
 add_mtext_dynamic_manual_height_columns () (*ezdxf.layouts.BaseLayout method*), 257
 add_mtext_static_columns () (*ezdxf.layouts.BaseLayout method*), 257
 add_multi_point_linear_dim () (*ezdxf.layouts.BaseLayout method*), 262
 add_multileader_block () (*ezdxf.layouts.BaseLayout method*), 272
 add_multileader_mtext () (*ezdxf.layouts.BaseLayout method*), 272
 add_new_dict () (*ezdxf.entities.Dictionary method*), 375
 add_open_spline () (*ezdxf.layouts.BaseLayout method*), 260
 add_ordinate_dim () (*ezdxf.layouts.BaseLayout method*), 271
 add_ordinate_x_dim () (*ezdxf.layouts.BaseLayout method*), 272
 add_ordinate_y_dim () (*ezdxf.layouts.BaseLayout method*), 272

add_placeholder () (*ezdxf.sections.objects.ObjectsSection method*), 215
 add_point () (*ezdxf.addons.r12writer.R12FastStreamWriter method*), 561
 add_point () (*ezdxf.layouts.BaseLayout method*), 253
 add_polyface () (*ezdxf.addons.r12writer.R12FastStreamWriter method*), 563
 add_polyface () (*ezdxf.layouts.BaseLayout method*), 256
 add_polyline () (*ezdxf.addons.r12writer.R12FastStreamWriter method*), 563
 add_polyline2d () (*ezdxf.layouts.BaseLayout method*), 255
 add_polyline3d () (*ezdxf.layouts.BaseLayout method*), 256
 add_polyline_2d () (*ezdxf.addons.r12writer.R12FastStreamWriter method*), 562
 add_polyline_path () (*ezdxf.entities.BoundaryPaths method*), 307
 add_polymesh () (*ezdxf.addons.r12writer.R12FastStreamWriter method*), 563
 add_polymesh () (*ezdxf.layouts.BaseLayout method*), 256
 add_radius_dim () (*ezdxf.layouts.BaseLayout method*), 263
 add_radius_dim_2p () (*ezdxf.layouts.BaseLayout method*), 264
 add_radius_dim_cra () (*ezdxf.layouts.BaseLayout method*), 265
 add_rational_spline () (*ezdxf.layouts.BaseLayout method*), 260
 add_ray () (*ezdxf.layouts.BaseLayout method*), 258
 add_region () (*ezdxf.layouts.BaseLayout method*), 273

add_required_classes () (*ezdxf.sections.classes.ClassesSection method*), 210
 add_revolved_surface () (*ezdxf.layouts.BaseLayout method*), 274
 add_shape () (*ezdxf.layouts.BaseLayout method*), 256
 add_shx () (*ezdxf.sections.table.TextstyleTable method*), 218
 add_solid () (*ezdxf.addons.r12writer.R12FastStreamWriter method*), 562
 add_solid () (*ezdxf.layouts.BaseLayout method*), 254
 add_spline () (*ezdxf.entities.EdgePath method*), 310
 add_spline () (*ezdxf.layouts.BaseLayout method*), 259
 add_spline () (*in module ezdxf.path*), 470
 add_spline_control_frame () (*ezdxf.layouts.BaseLayout method*), 259
 add_station () (*ezdxf.render.trace.LinearTrace*)

method), 538
add_surface() (*ezdxf.layouts.BaseLayout method*), 273
add_swept_surface() (*ezdxf.layouts.BaseLayout method*), 274
add_text() (*ezdxf.addons.r12writer.R12FastStreamWriter*), 111
add_text() (*ezdxf.layouts.BaseLayout method*), 254
add_to_layout() (*ezdxf.math.ConstructionArc method*), 440
add_to_layout() (*ezdxf.math.ConstructionEllipse method*), 444
add_trace() (*ezdxf.layouts.BaseLayout method*), 254
add_underlay() (*ezdxf.layouts.BaseLayout method*), 261
add_underlay_def() (*ezdxf.document.Drawing method*), 203
add_underlay_def() (*ezdxf.sections.objects.ObjectsSection method*), 215
add_vertices() (*ezdxf.render.MeshBuilder method*), 533
add_viewport() (*ezdxf.layouts.Paperspace method*), 278
add_wipeout() (*ezdxf.layouts.BaseLayout method*), 261
add_xline() (*ezdxf.layouts.BaseLayout method*), 258
add_xrecord() (*ezdxf.entities.Dictionary method*), 375
add_xrecord() (*ezdxf.entities.xdict.ExtensionDict method*), 391
add_xrecord() (*ezdxf.sections.objects.ObjectsSection method*), 215
add_xref_def() (*ezdxf.document.Drawing method*), 204
adjust_for_background (*ezdxf.entities.Underlay attribute*), 367
align_angle (*ezdxf.entities.ExtrudedSurface.dxf attribute*), 358
align_angle (*ezdxf.entities.SweptSurface.dxf attribute*), 360
align_direction (*ezdxf.entities.LoftedSurface.dxf attribute*), 359
align_point (*ezdxf.entities.Text.dxf attribute*), 361
align_space (*ezdxf.entities.MLeaderStyle.dxf attribute*), 381
align_start (*ezdxf.entities.ExtrudedSurface.dxf attribute*), 358
align_start (*ezdxf.entities.SweptSurface.dxf attribute*), 360
ALIGNED (*ezdxf.enums.TextEntityAlignment attribute*), 398
alignment (*ezdxf.entities.MTextData attribute*), 341
all_columns_plain_text() (*ezdxf.entities.MText method*), 333
all_columns_raw_content() (*ezdxf.entities.MText method*), 333
all_inside() (*ezdxf.math.BoundingBox method*), 433
all_lines_to_curve3() (*ezdxf.path.Path method*), 474
all_lines_to_curve4() (*ezdxf.path.Path method*), 474
all_to_line_edges() (*ezdxf.entities.BoundaryPaths method*), 307
all_to_spline_edges() (*ezdxf.entities.BoundaryPaths method*), 307
ambient_light_color_1 (*ezdxf.entities.Viewport.dxf attribute*), 372
ambient_light_color_2 (*ezdxf.entities.Viewport.dxf attribute*), 372
ambient_light_color_3 (*ezdxf.entities.Viewport.dxf attribute*), 372
angle (*ezdxf.entities.Dimension.dxf attribute*), 293
angle (*ezdxf.entities.PatternLine attribute*), 312
angle (*ezdxf.entities.Point.dxf attribute*), 343
angle (*ezdxf.math.ConstructionBox attribute*), 444
angle (*ezdxf.math.ConstructionRay attribute*), 435
angle (*ezdxf.math.Vec3 attribute*), 429
angle_about() (*ezdxf.math.Vec3 method*), 431
angle_between() (*ezdxf.math.Vec3 method*), 431
angle_deg (*ezdxf.math.ConstructionRay attribute*), 435
angle_deg (*ezdxf.math.Vec3 attribute*), 429
angle_span (*ezdxf.math.ConstructionArc attribute*), 439
angle_unit_name() (*in module ezdxf.units*), 38
angles() (*ezdxf.entities.Arc method*), 289
angles() (*ezdxf.math.ConstructionArc method*), 439
Angstroms (*ezdxf.enums.InsertUnits attribute*), 400
AngularUnits (*class in ezdxf.enums*), 401
annotation_handle (*ezdxf.entities.Leader.dxf attribute*), 316
annotation_type (*ezdxf.entities.Leader.dxf attribute*), 316
any_inside() (*ezdxf.math.BoundingBox method*), 433
any_inside() (*ezdxf.math.BoundingBox2d method*), 434
app_name (*ezdxf.entities.DXFClass.dxf attribute*), 210
AppData (*class in ezdxf.entities.appdata*), 697
appdata (*ezdxf.lldxr.extendedtags.ExtendedTags attribute*), 692
append() (*ezdxf.entities.LWPolyline method*), 321
append() (*ezdxf.entities.mline.ezdxf.entities.mline.MLineStyleElements method*), 325

append() (*ezdxf.entities.MText method*), 333
 append() (*ezdxf.lldxp.packedtags.VertexArray method*), 695
 append() (*ezdxf.math.Shape2d method*), 447
 append() (*ezdxf.render.Bezier method*), 526
 append() (*ezdxf.render.trace.TraceBuilder method*), 537
 append() (*ezdxf.sections.header.CustomVars method*), 210
 append() (*ezdxf.tools.text.MTextEditor method*), 492
 append_col() (*ezdxf.math.Matrix method*), 458
 append_face() (*ezdxf.entities.Polyface method*), 350
 append_faces() (*ezdxf.entities.Polyface method*), 350
 append_formatted_vertices() (*ezdxf.entities.Polyline method*), 347
 append_path() (*ezdxf.path.Path method*), 474
 append_points() (*ezdxf.entities.LWPolyline method*), 321
 append_reactor_handle() (*ezdxf.entities.DXFEntity method*), 284
 append_row() (*ezdxf.math.Matrix method*), 458
 append_vertex() (*ezdxf.entities.Polyline method*), 346
 append_vertices() (*ezdxf.entities.Polyline method*), 346
 AppID (*class in ezdxf.entities*), 238
 appids (*ezdxf.document.Drawing attribute*), 201
 appids (*ezdxf.sections.tables.TablesSection attribute*), 211
 AppIDTable (*class in ezdxf.sections.table*), 219
 apply() (*ezdxf.addons.geo.GeoProxy method*), 549
 apply_construction_tool() (*ezdxf.entities.Arc method*), 289
 apply_construction_tool() (*ezdxf.entities.Ellipse method*), 301
 apply_construction_tool() (*ezdxf.entities.Spline method*), 356
 apply_factor(*ezdxf.addons.acadctb.ColorDependentPlotStyle attribute*), 575
 apply_factor(*ezdxf.addons.acadctb.NamedPlotStyles attribute*), 576
 APPROXIMATE (*ezdxf.addons.drawing.config.LinePolicy attribute*), 543
 approximate() (*ezdxf.math.Bezier method*), 450
 approximate() (*ezdxf.math.Bezier3P method*), 452
 approximate() (*ezdxf.math.Bezier4P method*), 451
 approximate() (*ezdxf.math.BezierSurface method*), 454
 approximate() (*ezdxf.math.BSpline method*), 448
 approximate() (*ezdxf.math.EulerSpiral method*), 454
 approximate() (*ezdxf.path.Path method*), 474
 approximate() (*ezdxf.render.R12Spline method*), 525
 approximated_length() (*ezdxf.math.Bezier3P method*), 453
 approximated_length() (*ezdxf.math.Bezier4P method*), 452
 ApproxParamT (*class in ezdxf.math*), 453
 Arc (*class in ezdxf.entities*), 289
 ARC (*ezdxf.entities.EdgeType attribute*), 311
 arc_angle_span_deg() (*in module ezdxf.math*), 408
 arc_angle_span_rad() (*in module ezdxf.math*), 408
 arc_approximation() (*ezdxf.math.BSpline static method*), 449
 arc_chord_length() (*in module ezdxf.math*), 409
 arc_edges_to_ellipse_edges() (*ezdxf.entities.BoundaryPaths method*), 307
 arc_length_parameterization (*ezdxf.entities.LoftedSurface.dxf attribute*), 359
 arc_segment_count() (*in module ezdxf.math*), 408
 arc_to_bulge() (*in module ezdxf.math*), 410
 ArcDimension (*class in ezdxf.entities*), 299
 ArcEdge (*class in ezdxf.entities*), 311
 Architectural (*ezdxf.enums.LengthUnits attribute*), 401
 area() (*in module ezdxf.math*), 408
 arrow_head_handle (*ezdxf.entities.MLeaderStyle.dxf attribute*), 381
 arrow_head_handle (*ezdxf.entities.MultiLeader.dxf attribute*), 335
 arrow_head_size (*ezdxf.entities.MLeaderContext attribute*), 339
 arrow_head_size (*ezdxf.entities.MLeaderStyle.dxf attribute*), 381
 arrow_head_size (*ezdxf.entities.MultiLeader.dxf attribute*), 336
 arrow_heads (*ezdxf.entities.MultiLeader attribute*), 338
 HatchStyleHeadData (*class in ezdxf.entities*), 340
 ascending() (*in module ezdxf.reorder*), 511
 asdict() (*ezdxf.gfxattribs.GfxAttribs method*), 491
 aspect_ratio (*ezdxf.entities.VPort.dxf attribute*), 235
 associate() (*ezdxf.entities.Hatch method*), 305
 associative (*ezdxf.entities.Hatch.dxf attribute*), 302
 AstronomicalUnits (*ezdxf.enums.InsertUnits attribute*), 400
 AT_LEAST (*ezdxf.enums.MTextLineSpacing attribute*), 399
 attachment_direction (*ezdxf.entities.LeaderData attribute*), 340
 attachment_point (*ezdxf.entities.Dimension.dxf attribute*), 294
 attachment_point (*ezdxf.entities.MText.dxf attribute*), 331

attachment_type (*ezdxf.entities.MLeaderContext attribute*), 339
AttDef (*class in ezdxf.entities*), 248
attdefs () (*ezdxf.layouts.BlockLayout method*), 278
Attrib (*class in ezdxf.entities*), 247
AttribData (*class in ezdxf.entities*), 340
attribs (*ezdxf.entities.Insert attribute*), 243
audit () (*ezdxf.document.Drawing method*), 204
audit () (*ezdxf.entities.dxfgroups.DXFGroup method*), 280
audit () (*ezdxf.entities.dxfgroups.GroupCollection method*), 280
AUTOMATIC (*in module ezdxf.addons.acadctb*), 578
axis_point (*ezdxf.entities.RevolvedSurface.dxf attribute*), 359
axis_rotate () (*ezdxf.math.Matrix44 class method*), 425
axis_vector (*ezdxf.entities.RevolvedSurface.dxf attribute*), 359

B

back_clip_plane_z_value (*ezdxf.entities.Viewport.dxf attribute*), 369
back_clipping (*ezdxf.entities.View.dxf attribute*), 237
back_clipping (*ezdxf.entities.VPort.dxf attribute*), 235
background_handle (*ezdxf.entities.View.dxf attribute*), 238
background_handle (*ezdxf.entities.Viewport.dxf attribute*), 371
banded_matrix () (*in module ezdxf.math*), 457
BandedMatrixLU (*class in ezdxf.math*), 460
bank (*ezdxf.entities.ExtrudedSurface.dxf attribute*), 358
bank (*ezdxf.entities.SweptSurface.dxf attribute*), 360
base_point (*ezdxf.entities.Block.dxf attribute*), 241
base_point (*ezdxf.entities.MLeaderContext attribute*), 339
base_point (*ezdxf.entities.PatternLine attribute*), 313
base_point_set (*ezdxf.entities.ExtrudedSurface.dxf attribute*), 358
base_point_set (*ezdxf.entities.SweptSurface.dxf attribute*), 360
base_ucs_handle (*ezdxf.entities.View.dxf attribute*), 238
BaseLayout (*class in ezdxf.layouts*), 250
baseline (*ezdxf.tools.fonts.FontMeasurements attribute*), 502
baseline_vertices () (*ezdxf.tools.text.TextLine method*), 496
basic_transformation () (*in module ezdxf.math*), 415
bbox () (*in module ezdxf.path*), 469
best_fit_normal () (*in module ezdxf.math*), 415
Bezier (*class in ezdxf.math*), 450
Bezier (*class in ezdxf.render*), 526
Bezier3P (*class in ezdxf.math*), 452
Bezier4P (*class in ezdxf.math*), 451
bezier_decomposition () (*ezdxf.math.BSpline method*), 449
bezier_to_bspline () (*in module ezdxf.math*), 421
BezierSurface (*class in ezdxf.math*), 454
bg_color (*ezdxf.entities.MTextData attribute*), 341
bg_fill (*ezdxf.entities.MText.dxf attribute*), 331
bg_fill_color (*ezdxf.entities.MText.dxf attribute*), 332
bg_fill_color_name (*ezdxf.entities.MText.dxf attribute*), 332
bg_fill_true_color (*ezdxf.entities.MText.dxf attribute*), 332
bg_scale_factor (*ezdxf.entities.MTextData attribute*), 341
bg_transparency (*ezdxf.entities.MTextData attribute*), 342
bigfont (*ezdxf.entities.Textstyle.dxf attribute*), 226
BinaryRecord (*class in ezdxf.urecord*), 488
bisectrix () (*ezdxf.math.ConstructionRay method*), 435
BLACK (*ezdxf.enums.AC1 attribute*), 402
black () (*in module ezdxf.addons.dxf2code*), 554
black_code_str () (*ezdxf.addons.dxf2code.Code method*), 555
blend_cresce (*ezdxf.entities.Mesh.dxf attribute*), 326
Block (*class in ezdxf.entities*), 241
block (*ezdxf.entities.MLeaderContext attribute*), 339
block () (*ezdxf.entities.Insert method*), 243
block_attribs (*ezdxf.entities.MultiLeader attribute*), 338
block_color (*ezdxf.entities.Leader.dxf attribute*), 316
block_color (*ezdxf.entities.MLeaderStyle.dxf attribute*), 381
block_color (*ezdxf.entities.MultiLeader.dxf attribute*), 336
block_connection_type (*ezdxf.entities.MLeaderStyle.dxf attribute*), 381
block_connection_type (*ezdxf.entities.MultiLeader.dxf attribute*), 336
block_record_handle (*ezdxf.entities.BlockData attribute*), 342
block_record_handle (*ezdxf.entities.GeoData.dxf attribute*), 377
block_record_handle (*ezdxf.entities.MLeaderStyle.dxf attribute*), 381
block_record_handle (*ezdxf.entities.MultiLeader.dxf attribute*), 336

block_records (*ezdxf.sections.tables.TablesSection attribute*), 212
 block_rotation (*ezdxf.entities.MLeaderStyle.dxf attribute*), 381
 block_rotation (*ezdxf.entities.MultiLeader.dxf attribute*), 336
 block_scale_vector
 (*ezdxf.entities.MultiLeader.dxf attribute*), 336
 block_scale_x (*ezdxf.entities.MLeaderStyle.dxf attribute*), 381
 block_scale_y (*ezdxf.entities.MLeaderStyle.dxf attribute*), 381
 block_scale_z (*ezdxf.entities.MLeaderStyle.dxf attribute*), 382
 block_to_code ()
 (*in module ezdxf.addons.dxf2code*), 554
 BlockData (*class in ezdxf.entities*), 342
 BlockDefinitionIndex (*class in ezdxf.blkrefs*), 392
 BlockLayout (*class in ezdxf.layouts*), 278
 BlockRecord (*class in ezdxf.entities*), 240
 BlockRecordTable (*class in ezdxf.sections.table*), 220
 BlockReferenceCounter (*class in ezdxf.blkrefs*), 392
 blocks (*ezdxf.addons.dxf2code.Code attribute*), 554
 blocks (*ezdxf.document.Drawing attribute*), 200
 BlocksSection (*class in ezdxf.sections.blocks*), 212
 BLUE (*ezdxf.enums.ACI attribute*), 402
 Body (*class in ezdxf.entities*), 290
 border_lines ()
 (*ezdxf.math.ConstructionBox method*), 445
 BOTTOM (*ezdxf.enums.MTextLineAlignment attribute*), 399
 bottom_attachment (*ezdxf.entities.MLeaderContext attribute*), 340
 BOTTOM_CENTER (*ezdxf.enums.MTextEntityAlignment attribute*), 398
 BOTTOM_CENTER (*ezdxf.enums.TextEntityAlignment attribute*), 398
 BOTTOM_LEFT (*ezdxf.enums.MTextEntityAlignment attribute*), 398
 BOTTOM_LEFT (*ezdxf.enums.TextEntityAlignment attribute*), 398
 BOTTOM_RIGHT (*ezdxf.enums.MTextEntityAlignment attribute*), 398
 BOTTOM_RIGHT (*ezdxf.enums.TextEntityAlignment attribute*), 398
 boundary_path (*ezdxf.entities.Image attribute*), 315
 boundary_path (*ezdxf.entities.Underlay attribute*), 367
 boundary_path_wcs ()
 (*ezdxf.entities.Image method*), 315
 BoundaryPaths (*class in ezdxf.entities*), 306
 BoundaryPathType (*class in ezdxf.entities*), 307
 bounding_box
 (*ezdxf.math.ConstructionArc attribute*), 439
 bounding_box
 (*ezdxf.math.ConstructionBox attribute*), 445
 bounding_box
 (*ezdxf.math.ConstructionCircle attribute*), 437
 bounding_box
 (*ezdxf.math.ConstructionLine attribute*), 436
 bounding_box
 (*ezdxf.math.Shape2d attribute*), 447
 BoundingBox (*class in ezdxf.math*), 432
 BoundingBox2d (*class in ezdxf.math*), 433
 box ()
 (*in module ezdxf.render.forms*), 529
 box_fill_scale (*ezdxf.entities.MText.dxf attribute*), 332
 break_gap_size (*ezdxf.entities.MLeaderStyle.dxf attribute*), 382
 breaks (*ezdxf.entities.LeaderData attribute*), 340
 breaks (*ezdxf.entities.LeaderLine attribute*), 340
 brightness (*ezdxf.entities.Image.dxf attribute*), 314
 BSpline (*class in ezdxf.math*), 448
 bspline ()
 (*ezdxf.math.EulerSpiral method*), 454
 build_system_font_cache ()
 (*in module ezdxf.tools.fonts*), 502
 bulge, 699
 bulge (*ezdxf.entities.Vertex.dxf attribute*), 348
 bulge_3_points ()
 (*in module ezdxf.math*), 410
 bulge_center ()
 (*in module ezdxf.math*), 409
 bulge_radius ()
 (*in module ezdxf.math*), 409
 bulge_to_arc ()
 (*in module ezdxf.math*), 410
 bullet_list ()
 (*ezdxf.tools.text.MTextEditor method*), 494
 by_handle ()
 (*ezdxf.blkrefs.BlockDefinitionIndex method*), 392
 by_handle ()
 (*ezdxf.blkrefs.BlockReferenceCounter method*), 392
 by_name ()
 (*ezdxf.blkrefs.BlockDefinitionIndex method*), 392
 by_name ()
 (*ezdxf.blkrefs.BlockReferenceCounter method*), 392
 BY_STYLE (*ezdxf.enums.MTextFlowDirection attribute*), 399
 BYBLOCK (*ezdxf.enums.ACI attribute*), 401
 BYLAYER (*ezdxf.enums.ACI attribute*), 402
 BYOBJECT (*ezdxf.enums.ACI attribute*), 402
 bytes_to_hexstr ()
 (*in module ezdxf.tools*), 489

C

Cache (*class in ezdxf.bbox*), 479
 CAD, 699
 calendardate ()
 (*in module ezdxf.tools*), 489
 camera_plottable
 (*ezdxf.entities.View.dxf attribute*), 238

CANVAS (*ezdxf.enums.MTextBackgroundColor attribute*), 399
cap_height (*ezdxf.tools.fonts.FontMeasurements attribute*), 502
cap_height (*ezdxf.tools.text_size.ezdxftools.text_size.TextSizes attribute*), 499
caret_decode () (*in module ezdxf.tools.text*), 497
ccw (*ezdxf.entities.ArcEdge attribute*), 311
ccw (*ezdxf.entities.EllipseEdge attribute*), 311
center (*ezdxf.entities.Arc.dxf attribute*), 289
center (*ezdxf.entities.ArcEdge attribute*), 311
center (*ezdxf.entities.Circle.dxf attribute*), 291
center (*ezdxf.entities.Ellipse.dxf attribute*), 300
center (*ezdxf.entities.Viewport.dxf attribute*), 368
center (*ezdxf.entities.VPort.dxf attribute*), 235
CENTER (*ezdxf.enums.MTextParagraphAlignment attribute*), 398
CENTER (*ezdxf.enums.TextEntityAlignment attribute*), 397
center (*ezdxf.math.ConstructionArc attribute*), 439
center (*ezdxf.math.ConstructionBox attribute*), 444
center (*ezdxf.math.ConstructionCircle attribute*), 437
center (*ezdxf.math.ConstructionEllipse attribute*), 442
CENTER (*ezdxf.tools.text.ezdxflldxf.const.MTextParagraphAlignment attribute*), 495
center () (*in module ezdxf.zoom*), 510
center_point (*ezdxf.entities.View.dxf attribute*), 237
centered (*ezdxf.entities.Gradient attribute*), 313
Centimeters (*ezdxf.enums.InsertUnits attribute*), 400
chain () (*ezdxf.math.Matrix44 static method*), 426
chain_layouts_and_blocks ()
 (*ezdxf.document.Drawing method*), 204
char_height (*ezdxf.entities.MLeaderContext attribute*), 339
char_height (*ezdxf.entities.MLeaderStyle.dxf attribute*), 382
char_height (*ezdxf.entities.MText.dxf attribute*), 330
char_tracking_factor ()
 (*ezdxf.tools.text.MTextEditor method*), 493
Circle (*class in ezdxf.entities*), 291
circle () (*in module ezdxf.render.forms*), 528
circle_approximation_count
 (*ezdxf.addons.drawing.config.Configuration attribute*), 542
circle_center () (*ezdxf.math.EulerSpiral method*), 454
circle_zoom (*ezdxf.entities.Viewport.dxf attribute*), 369
circle_zoom (*ezdxf.entities.VPort.dxf attribute*), 236
circumcircle_radius
 (*ezdxf.math.ConstructionBox attribute*), 445
class_id (*ezdxf.entities.ExtrudedSurface.dxf attribute*), 358
class_id (*ezdxf.entities.RevolvedSurface.dxf attribute*), 359
class_version (*ezdxf.entities.ImageDef.dxf attribute*), 380
class_version (*ezdxf.entities.ImageDefReactor.dxf attribute*), 380
classes (*ezdxf.document.Drawing attribute*), 200
classes (*ezdxf.sections.classes.ClassesSection attribute*), 210
ClassesSection (*class in ezdxf.sections.classes*), 210
clear () (*ezdxf.entities.BoundaryPaths method*), 307
clear () (*ezdxf.entities.Dictionary method*), 375
clear () (*ezdxf.entities.dxfgroups.DXFGroup method*), 279
clear () (*ezdxf.entities.dxfgroups.GroupCollection method*), 280
clear () (*ezdxf.entities.EdgePath method*), 309
clear () (*ezdxf.entities.LWPolyline method*), 321
clear () (*ezdxf.entities.MLine method*), 324
clear () (*ezdxf.entities.Pattern method*), 312
clear () (*ezdxf.entities.PolylinePath method*), 308
clear () (*ezdxf.entities.XRecord method*), 388
clear () (*ezdxf.entitydb.EntitySpace method*), 688
clear () (*ezdxf.lldxflldxf.packedtags.TagList method*), 694
clear () (*ezdxf.lldxflldxf.packedtags.VertexArray method*), 695
clear () (*ezdxf.sections.header.CustomVars method*), 209
clear () (*ezdxf.tools.text.MTextEditor method*), 493
clip_mode (*ezdxf.entities.Image.dxf attribute*), 314
clip_polygon_2d () (*in module ezdxf.math*), 412
clipping (*ezdxf.entities.Image.dxf attribute*), 314
clipping (*ezdxf.entities.Underlay attribute*), 367
clipping_boundary_handle
 (*ezdxf.entities.Viewport.dxf attribute*), 370
clipping_boundary_type
 (*ezdxf.entities.Image.dxf attribute*), 314
clockwise () (*ezdxf.path.Path method*), 473
clone () (*ezdxf.lldxflldxf.extendedtags.ExtendedTags method*), 692
clone () (*ezdxf.lldxflldxf.packedtags.TagList method*), 694
clone () (*ezdxf.lldxflldxf.packedtags.VertexArray method*), 695
clone () (*ezdxf.lldxftypes.DXFTag method*), 689
clone () (*ezdxf.path.Path method*), 473
cloning (*ezdxf.entities.Dictionary.dxf attribute*), 374
cloning (*ezdxf.entities.XRecord.dxf attribute*), 387
close () (*ezdxf.addons.iterdxfl.IterDXF method*), 558
close () (*ezdxf.addons.iterdxfl.IterDXFWriter method*), 558
close () (*ezdxf.addons.r12writer.R12FastStreamWriter method*), 560
close () (*ezdxf.entities.LWPolyline method*), 320
close () (*ezdxf.entities.MLine method*), 323

close() (*ezdxf.entities.Polyline* method), 346
close() (*ezdxf.path.Path* method), 473
close() (*ezdxf.render.trace.TraceBuilder* method), 537
close_sub_path() (*ezdxf.path.Path* method), 473
close_to_axis (*ezdxf.entities.RevolvedSurface.dxf_attribute*), 359
closed (*ezdxf.entities.Spline* attribute), 356
closed_surfaces (*ezdxf.entities.LoftedSurface.dxf_attribute*), 359
closed_uniform_bspline() (in module *ezdxf.math*), 420
closest_point() (in module *ezdxf.math*), 407
Code (class in *ezdxf.addons.dxf2code*), 554
code (*ezdxf.addons.dxf2code.Code* attribute), 554
code (*ezdxf.lldx.types.DXFTag* attribute), 689
code_str() (*ezdxf.addons.dxf2code.Code* method), 555
col() (*ezdxf.math.Matrix* method), 458
collect_consecutive_tags() (*ezdxf.lldx.tags.Tags* method), 691
color (*ezdxf.entities.BlockData* attribute), 342
color (*ezdxf.entities.DXFGraphic.dxf* attribute), 286
color (*ezdxf.entities.ezdx.entities.mline.MLineStyleElement* attribute), 325
color (*ezdxf.entities.Layer* attribute), 222
color (*ezdxf.entities.Layer.dxf* attribute), 221
color (*ezdxf.entities.LeaderLine* attribute), 340
color (*ezdxf.entities.MTextData* attribute), 341
color (*ezdxf.entities.Sun.dxf* attribute), 385
COLOR (*ezdxf.enums.MTextBackgroundColor* attribute), 399
color() (*ezdxf.tools.text.MTextEditor* method), 493
color1 (*ezdxf.entities.Gradient* attribute), 313
color2 (*ezdxf.entities.Gradient* attribute), 313
color_name (*ezdxf.entities.DXFGraphic.dxf* attribute), 286
ColorDependentPlotStyles (class in *ezdxf.addons.acadctb*), 575
cols() (*ezdxf.math.Matrix* method), 458
column_count (*ezdxf.entities.Insert.dxf* attribute), 243
column_count (*ezdxf.tools.text_size.ezdx.tools.text_size.MTextSize* attribute), 499
column_flow_reversed (*ezdxf.entities.MTextData* attribute), 342
column_gutter_width (*ezdxf.entities.MTextData* attribute), 342
column_heights (*ezdxf.tools.text_size.ezdx.tools.text_size.MTextSize* attribute), 499
column_sizes (*ezdxf.entities.MTextData* attribute), 342
column_spacing (*ezdxf.entities.Insert.dxf* attribute), 243
column_type (*ezdxf.entities.MTextData* attribute), 342
column_width (*ezdxf.entities.MTextData* attribute), 342
column_width (*ezdxf.tools.text_size.ezdx.tools.text_size.MTextSize* attribute), 499
columns() (*ezdxf.math.Matrix44* method), 427
commit() (*ezdxf.entities.DimStyleOverride* method), 296
commit() (*ezdxf.entities.xdata.XDataUserDict* method), 486
commit() (*ezdxf.entities.xdata.XDataUserList* method), 485
commit() (*ezdxf.urecord.BinaryRecord* method), 488
commit() (*ezdxf.urecord.UserRecord* method), 488
compact_banded_matrix() (in module *ezdxf.math*), 457
cone() (in module *ezdxf.render.forms*), 531
cone_2p() (in module *ezdxf.render.forms*), 531
Configuration (class in *ezdxf.addons.drawing.config*), 541
const_width (*ezdxf.entities.LWPolyline.dxf* attribute), 320
construction_tool() (*ezdxf.entities.Arc* method), 289
construction_tool() (*ezdxf.entities.Ellipse* method), 301
construction_tool() (*ezdxf.entities.Spline* method), 356
ConstructionArc (class in *ezdxf.math*), 438
ConstructionBox (class in *ezdxf.math*), 444
ConstructionCircle (class in *ezdxf.math*), 436
ConstructionEllipse (class in *ezdxf.math*), 442
ConstructionLine (class in *ezdxf.math*), 435
ConstructionPolyline (class in *ezdxf.math*), 446
ConstructionRay (class in *ezdxf.math*), 434
contains() (*ezdxf.math.BoundingBox* method), 433
contains() (*ezdxf.math.BoundingBox2d* method), 434
content_type (*ezdxf.entities.MLeaderStyle.dxf* attribute), 382
content_type (*ezdxf.entities.MultiLeader.dxf* attribute), 336
context (*ezdxf.entities.MultiLeader* attribute), 338
MTextSize (*ezdxf.entities.Image.dxf* attribute), 314
contrast (*ezdxf.entities.Underlay.dxf* attribute), 367
control_point_count() (*ezdxf.entities.Spline* method), 356
control_point_tolerance (*ezdxf.entities.Spline.dxf* attribute), 355
control_points (*ezdxf.entities.Spline* attribute), 356
control_points (*ezdxf.entities.SplineEdge* attribute), 312
control_points (*ezdxf.math.Bezier* attribute), 450
control_points (*ezdxf.math.Bezier3P* attribute), 452
control_points (*ezdxf.math.Bezier4P* attribute), 451

control_vertices () (*ezdxf.path.Path method*), 473
conversion_factor () (*in module ezdxf.units*), 38
convex_hull () (*ezdxf.math.Shape2d method*), 447
convex_hull_2d () (*in module ezdxf.math*), 411
coordinate_projection_radius
 (*ezdxf.entities.GeoData.dxf attribute*), 378
coordinate_system_definition
 (*ezdxf.entities.GeoData attribute*), 378
coordinate_type (*ezdxf.entities.GeoData.dxf attribute*), 377
copy () (*ezdxf.addons.geo.GeoProxy method*), 548
copy () (*ezdxf.math.Matrix44 method*), 425
copy () (*ezdxf.math.Plane method*), 432
copy () (*ezdxf.math.UCS method*), 422
copy () (*ezdxf.math.Vec3 method*), 429
copy () (*ezdxf.render.MeshBuilder method*), 533
copy_to_header () (*ezdxf.entities.DimStyle method*), 232
copy_to_layout () (*ezdxf.entities.DXFGraphic method*), 284
corner_vertices () (*ezdxf.tools.text.TextLine method*), 496
corners (*ezdxf.math.ConstructionBox attribute*), 444
count (*ezdxf.entities.LWPolyline.dxf attribute*), 320
count (*ezdxf.entities.MLine.dxf attribute*), 323
count () (*ezdxf.entities.Dictionary method*), 375
count_boundary_points (*ezdxf.entities.Image.dxf attribute*), 314
counter_clockwise () (*ezdxf.path.Path method*), 474
cpp_class_name (*ezdxf.entities.DXFClass.dxf attribute*), 210
creases (*ezdxf.entities.Mesh attribute*), 326
cross () (*ezdxf.math.Vec3 method*), 431
crs_to_wcs () (*ezdxf.addons.geo.GeoProxy method*), 548
CSG (*class in ezdxf.addons.pycsg*), 573
CTB, 699
cube () (*in module ezdxf.render.forms*), 530
cube_vertices () (*ezdxf.math.BoundingBox method*), 433
cubes () (*ezdxf.addons.MengerSponge method*), 583
cubic_bezier_approximation ()
 (*ezdxf.math.BSpline method*), 450
cubic_bezier_from_3p () (*in module ezdxf.math*), 420
cubic_bezier_from_arc () (*in module ezdxf.math*), 420
cubic_bezier_from_ellipse () (*in module ezdxf.math*), 420
cubic_bezier_interpolation () (*in module ezdxf.math*), 420
curve3_to () (*ezdxf.path.Path method*), 473
curve4_to () (*ezdxf.path.Path method*), 473
CurvedTrace (*class in ezdxf.render.trace*), 538
custom_lineweight_display_units
 (*ezdxf.addons.acadctb.ColorDependentPlotStyles attribute*), 575
custom_lineweight_display_units
 (*ezdxf.addons.acadctb.NamedPlotStyles attribute*), 576
custom_vars (*ezdxf.sections.header.HeaderSection attribute*), 209
CustomVars (*class in ezdxf.sections.header*), 209
CYAN (*ezdxf.enums.ACI attribute*), 402
cylinder () (*in module ezdxf.render.forms*), 530
cylinder_2p () (*in module ezdxf.render.forms*), 531

D

dash_length_items (*ezdxf.entities.PatternLine attribute*), 313
data (*ezdxf.urecord.BinaryRecord attribute*), 488
data (*ezdxf.urecord.UserRecord attribute*), 487
data () (*ezdxf.math.ConstructionPolyline method*), 446
daylight_savings_time (*ezdxf.entities.Sun.dxf attribute*), 385
dd2dms () (*in module ezdxf.addons.geo*), 549
Decameters (*ezdxf.enums.InsertUnits attribute*), 400
Decimal (*ezdxf.enums.LengthUnits attribute*), 401
DecimalDegrees (*ezdxf.enums.AngularUnits attribute*), 401
Decimeters (*ezdxf.enums.InsertUnits attribute*), 400
decode () (*in module ezdxf.tools.crypt*), 490
decode_base64 () (*in module ezdxf*), 197
decode_dxf_unicode () (*in module ezdxf*), 489
decode_raw_color () (*in module ezdxf.colors*), 402
decode_raw_color_int () (*in module ezdxf.colors*), 402
default (*ezdxf.entities.DictionaryWithDefault.dxf attribute*), 376
DEFAULT (*ezdxf.enums.MTextParagraphAlignment attribute*), 398
DEFAULT (*ezdxf.tools.text.ezdxflldxf.const.MTextParagraphAlignment attribute*), 495
default_content (*ezdxf.entities.MTextData attribute*), 341
default_dimension_text_style (*in module ezdxf.options*), 506
default_end_width (*ezdxf.entities.Polyline.dxf attribute*), 345
default_lighting_flag
 (*ezdxf.entities.Viewport.dxf attribute*), 372
default_lighting_style
 (*ezdxf.entities.Viewport.dxf attribute*), 372
default_paths () (*ezdxf.entities.BoundaryPaths method*), 306
default_start_width (*ezdxf.entities.Polyline.dxf attribute*), 345

default_text_content
 (ezdxf.entities.MLeaderStyle.dxf attribute), 382
defaults() (ezdxf.addons.drawing.config.Configuration method), 542
defined_height (ezdxf.entities.MTextData attribute), 341
defpoint (ezdxf.entities.Dimension.dxf attribute), 293
defpoint2 (ezdxf.entities.ArcDimension.dxf attribute), 300
defpoint2 (ezdxf.entities.Dimension.dxf attribute), 293
defpoint3 (ezdxf.entities.ArcDimension.dxf attribute), 300
defpoint3 (ezdxf.entities.Dimension.dxf attribute), 293
defpoint4 (ezdxf.entities.ArcDimension.dxf attribute), 300
defpoint4 (ezdxf.entities.Dimension.dxf attribute), 293
defpoint5 (ezdxf.entities.Dimension.dxf attribute), 293
degree (ezdxf.entities.Spline.dxf attribute), 355
degree (ezdxf.entities.SplineEdge attribute), 312
DegreesMinutesSeconds
 (ezdxf.enumsAngularUnits attribute), 401
del_dxf_attrib() (ezdxf.entities.DXFEntity method), 282
delete() (ezdxf.entities.dxfgroups.GroupCollection method), 280
delete() (ezdxf.layouts.Layouts method), 250
delete_all_attribs() (ezdxf.entities.Insert method), 245
delete_all_blocks()
 (ezdxf.sections.blocks.BlocksSection method), 213
delete_all_entities()
 (ezdxf.layouts.BaseLayout method), 251
delete_attrib() (ezdxf.entities.Insert method), 245
delete_block() (ezdxf.sections.blocks.BlocksSection method), 212
delete_config() (ezdxf.sections.table.ViewportTable method), 220
delete_default_config_files() (in module ezdxf.options), 506
delete_entity() (ezdxf.layouts.BaseLayout method), 251
delete_layout() (ezdxf.document.Drawing method), 202
derivative() (ezdxf.math.Bezier method), 451
derivative() (ezdxf.math.BSpline method), 449
derivatives() (ezdxf.math.Bezier method), 451
derivatives() (ezdxf.math.BSpline method), 449
descender_height (ezdxf.tools.fonts.FontMeasurements attribute), 502
descending() (in module ezdxf.reorder), 511
description (ezdxf.addons.acadctb.ColorDependentPlotStyles attribute), 575
description (ezdxf.addons.acadctb.NamedPlotStyles attribute), 576
description (ezdxf.addons.acadctb.PlotStyle attribute), 577
description (ezdxf.entities.dxfgroups.DXFGROUP.dxf attribute), 279
description (ezdxf.entities.Layer attribute), 222
description (ezdxf.entities.Linetype.dxf attribute), 226
description (ezdxf.entities.MLineStyle.dxf attribute), 324
design_point (ezdxf.entities.GeoData.dxf attribute), 377
destroy() (ezdxf.entities.xdict.ExtensionDict method), 391
detect_banded_matrix() (in module ezdxf.math), 457
determinant() (ezdxf.math.BandedMatrixLU method), 460
determinant() (ezdxf.math.LUdecomposition method), 460
determinant() (ezdxf.math.Matrix method), 459
determinant() (ezdxf.math.Matrix44 method), 427
DgnDefinition (class in ezdxf.entities), 387
DgnUnderlay (class in ezdxf.entities), 368
diag() (ezdxf.math.Matrix method), 458
Dictionary (class in ezdxf.entities), 374
DictionaryVar (class in ezdxf.entities), 376
DictionaryWithDefault (class in ezdxf.entities), 376
dimadec (ezdxf.entities.DimStyle.dxf attribute), 229
dimalt (ezdxf.entities.DimStyle.dxf attribute), 229
dimaltd (ezdxf.entities.DimStyle.dxf attribute), 229
dimaltf (ezdxf.entities.DimStyle.dxf attribute), 228
dimaltrnd (ezdxf.entities.DimStyle.dxf attribute), 228
dimaltd (ezdxf.entities.DimStyle.dxf attribute), 230
dimalttz (ezdxf.entities.DimStyle.dxf attribute), 231
dimaltu (ezdxf.entities.DimStyle.dxf attribute), 230
dimaltz (ezdxf.entities.DimStyle.dxf attribute), 231
dimapost (ezdxf.entities.DimStyle.dxf attribute), 227
dimarcosym (ezdxf.entities.DimStyle.dxf attribute), 232
dimasz (ezdxf.entities.DimStyle.dxf attribute), 227
dimatfit (ezdxf.entities.DimStyle.dxf attribute), 231
dimaunit (ezdxf.entities.DimStyle.dxf attribute), 230
dimazin (ezdxf.entities.DimStyle.dxf attribute), 229
dimblk (ezdxf.entities.DimStyle.dxf attribute), 227
dimblk1 (ezdxf.entities.DimStyle.dxf attribute), 227
dimblk1_handle (ezdxf.entities.DimStyle.dxf attribute), 231
dimblk2 (ezdxf.entities.DimStyle.dxf attribute), 227

dimblk2_handle (*ezdxf.entities.DimStyle.dxf attribute*), 231
dimblk_handle (*ezdxf.entities.DimStyle.dxf attribute*), 231
dimcen (*ezdxf.entities.DimStyle.dxf attribute*), 228
dimclrd (*ezdxf.entities.DimStyle.dxf attribute*), 229
dimclre (*ezdxf.entities.DimStyle.dxf attribute*), 229
dimclrt (*ezdxf.entities.DimStyle.dxf attribute*), 229
dimdec (*ezdxf.entities.DimStyle.dxf attribute*), 230
dimdle (*ezdxf.entities.DimStyle.dxf attribute*), 228
dimdli (*ezdxf.entities.DimStyle.dxf attribute*), 227
dimdsep (*ezdxf.entities.DimStyle.dxf attribute*), 230
Dimension (*class in ezdxf.entities*), 292
dimension (*ezdxf.entities.DimStyleOverride attribute*), 295
dimexe (*ezdxf.entities.DimStyle.dxf attribute*), 227
dimexo (*ezdxf.entities.DimStyle.dxf attribute*), 227
dimfit (*ezdxf.entities.DimStyle.dxf attribute*), 231
dimfrac (*ezdxf.entities.DimStyle.dxf attribute*), 230
dimfxl (*ezdxf.entities.DimStyle.dxf attribute*), 232
dimfxlon (*ezdxf.entities.DimStyle.dxf attribute*), 232
dimgap (*ezdxf.entities.DimStyle.dxf attribute*), 228
dimjust (*ezdxf.entities.DimStyle.dxf attribute*), 230
dimldrblk (*ezdxf.entities.DimStyle.dxf attribute*), 231
dimldrblk_handle (*ezdxf.entities.DimStyle.dxf attribute*), 231
dimlex1_handle (*ezdxf.entities.DimStyle.dxf attribute*), 231
dimlex2_handle (*ezdxf.entities.DimStyle.dxf attribute*), 232
dimlfac (*ezdxf.entities.DimStyle.dxf attribute*), 228
dimlim (*ezdxf.entities.DimStyle.dxf attribute*), 228
dimltx1 (*ezdxf.entities.DimStyle.dxf attribute*), 231
dimltx2 (*ezdxf.entities.DimStyle.dxf attribute*), 232
dimltype (*ezdxf.entities.DimStyle.dxf attribute*), 231
dimltype_handle (*ezdxf.entities.DimStyle.dxf attribute*), 231
dimlunit (*ezdxf.entities.DimStyle.dxf attribute*), 230
dimlwd (*ezdxf.entities.DimStyle.dxf attribute*), 231
dimlwe (*ezdxf.entities.DimStyle.dxf attribute*), 231
dimpost (*ezdxf.entities.DimStyle.dxf attribute*), 227
dimrnd (*ezdxf.entities.DimStyle.dxf attribute*), 227
dimsah (*ezdxf.entities.DimStyle.dxf attribute*), 229
dimscale (*ezdxf.entities.DimStyle.dxf attribute*), 227
dimsd1 (*ezdxf.entities.DimStyle.dxf attribute*), 230
dimsd2 (*ezdxf.entities.DimStyle.dxf attribute*), 230
dimse1 (*ezdxf.entities.DimStyle.dxf attribute*), 228
dimse2 (*ezdxf.entities.DimStyle.dxf attribute*), 228
dimsoxd (*ezdxf.entities.DimStyle.dxf attribute*), 229
DimStyle (*class in ezdxf.entities*), 227
dimstyle (*ezdxf.entities.Dimension.dxf attribute*), 292
dimstyle (*ezdxf.entities.DimStyleOverride attribute*), 295
dimstyle (*ezdxf.entities.Leader.dxf attribute*), 315
at-
dimstyle_attribs (*ezdxf.entities.DimStyleOverride attribute*), 295
DimStyleOverride (*class in ezdxf.entities*), 295
dimstyles (*ezdxf.addons.dxf2code.Code attribute*), 554
dimstyles (*ezdxf.document.Drawing attribute*), 200
dimstyles (*ezdxf.sections.tables.TablesSection attribute*), 211
DimStyleTable (*class in ezdxf.sections.table*), 219
dimtad (*ezdxf.entities.DimStyle.dxf attribute*), 228
dimtdec (*ezdxf.entities.DimStyle.dxf attribute*), 230
dimtfac (*ezdxf.entities.DimStyle.dxf attribute*), 228
dimtfill (*ezdxf.entities.DimStyle.dxf attribute*), 232
dimtfillclr (*ezdxf.entities.DimStyle.dxf attribute*), 232
dimtih (*ezdxf.entities.DimStyle.dxf attribute*), 228
dimtix (*ezdxf.entities.DimStyle.dxf attribute*), 229
dimtm (*ezdxf.entities.DimStyle.dxf attribute*), 228
dimtmove (*ezdxf.entities.DimStyle.dxf attribute*), 230
dimtofl (*ezdxf.entities.DimStyle.dxf attribute*), 229
dimtob (*ezdxf.entities.DimStyle.dxf attribute*), 228
dimtol (*ezdxf.entities.DimStyle.dxf attribute*), 228
dimtolj (*ezdxf.entities.DimStyle.dxf attribute*), 230
dimtp (*ezdxf.entities.DimStyle.dxf attribute*), 228
dimtsz (*ezdxf.entities.DimStyle.dxf attribute*), 228
dimtvp (*ezdxf.entities.DimStyle.dxf attribute*), 228
dimtxsty (*ezdxf.entities.DimStyle.dxf attribute*), 231
dimtxsty_handle (*ezdxf.entities.DimStyle.dxf attribute*), 231
dimtxt (*ezdxf.entities.DimStyle.dxf attribute*), 228
dimtype (*ezdxf.entities.ArcDimension attribute*), 300
dimtype (*ezdxf.entities.Dimension.dxf attribute*), 292
dimtzin (*ezdxf.entities.DimStyle.dxf attribute*), 230
dimunit (*ezdxf.entities.DimStyle.dxf attribute*), 229
dimupt (*ezdxf.entities.DimStyle.dxf attribute*), 231
dimzin (*ezdxf.entities.DimStyle.dxf attribute*), 229
direction (*ezdxf.math.ConstructionRay attribute*), 435
direction_from_wcs () (*ezdxf.math.ucs method*), 422
direction_point (*ezdxf.entities.View.dxf attribute*), 237
direction_point (*ezdxf.entities.VPort.dxf attribute*), 235
direction_to_wcs () (*ezdxf.math.ucs method*), 422
DISABLE (*ezdxf.enums.SortEntities attribute*), 401
disable_c_ext (*in module ezdxf.options*), 508
discard () (*ezdxf.document.ezdxf.document.MetaData.MetaData method*), 199
discard () (*ezdxf.entities.appdata.AppData method*), 697
discard () (*ezdxf.entities.appdata.Reactors method*), 698

discard() (*ezdxf.entities.Dictionary method*), 375
discard() (*ezdxf.entities.xdata.XData method*), 696
discard() (*ezdxf.entities.xdata.XDataUserDict method*), 486
discard() (*ezdxf.entities.xdict.ExtensionDict method*), 390
discard_app_data() (*ezdxf.entities.DXFEntity method*), 282
discard_extended_font_data() (*ezdxf.entities.Textstyle method*), 226
discard_extension_dict() (*ezdxf.entities.DXFEntity method*), 282
discard_reactor_handle() (*ezdxf.entities.DXFEntity method*), 284
discard_xdata() (*ezdxf.entities.DXFEntity method*), 283
discard_xdata_list() (*ezdxf.entities.DXFEntity method*), 283
discard_xlist() (*ezdxf.entities.xdata.XData method*), 696
distance() (*ezdxf.math.ApproxParamT method*), 453
distance() (*ezdxf.math.EulerSpiral method*), 454
distance() (*ezdxf.math.Vec3 method*), 431
distance_from_origin (*ezdxf.math.Plane attribute*), 432
distance_point_line_2d() (*in module ezdxf.math*), 410
distance_point_line_3d() (*in module ezdxf.math*), 416
distance_to() (*ezdxf.math.Plane method*), 432
DISTRIBUTED (*ezdxf.enums.MTextParagraphAlignment attribute*), 398
DISTRIBUTED (*ezdxf.tools.text.ezdxflldxf.const.MTextParagraphAlignment attribute*), 495
dithering (*ezdxf.addons.acadctb.PlotStyle attribute*), 578
divide() (*ezdxf.math.ConstructionPolyline method*), 446
divide_by_length() (*ezdxf.math.ConstructionPolyline method*), 446
dms2dd() (*in module ezdxf.addons.geo*), 549
doc (*ezdxf.entities.DXFEntity attribute*), 281
dogleg_length (*ezdxf.entities.LeaderData attribute*), 340
dogleg_length (*ezdxf.entities.MLeaderStyle.dxf attribute*), 382
dogleg_length (*ezdxf.entities.MultiLeader.dxf attribute*), 336
dogleg_vector (*ezdxf.entities.LeaderData attribute*), 340
dot() (*ezdxf.math.Vec3 method*), 431
draft_angle (*ezdxf.entities.ExtrudedSurface.dxf attribute*), 358
draft_angle (*ezdxf.entities.RevolvedSurface.dxf attribute*), 359
draft_angle (*ezdxf.entities.SweptSurface.dxf attribute*), 360
draft_end_distance (*ezdxf.entities.ExtrudedSurface.dxf attribute*), 358
draft_end_distance (*ezdxf.entities.SweptSurface.dxf attribute*), 360
draft_start_distance (*ezdxf.entities.ExtrudedSurface.dxf attribute*), 358
draft_start_distance (*ezdxf.entities.SweptSurface attribute*), 360
draw_leader_order_type (*ezdxf.entities.MLeaderStyle.dxf attribute*), 382
draw_mleader_order_type (*ezdxf.entities.MLeaderStyle.dxf attribute*), 382
draw_viewports_first() (*ezdxf.layouts.Layout method*), 276
Drawing (*class in ezdxf.document*), 199
DTYPE (*ezdxf.lldxflldxf.packedtags.TagArray attribute*), 694
duplicate_entry() (*ezdxf.sections.table.Table method*), 216
DwfDefinition (*class in ezdxf.entities*), 386
DwfUnderlay (*class in ezdxf.entities*), 368
DWG, 700
DXF, 700
dx (*ezdxf.entities.DXFEntity attribute*), 281
dx (*ezdxf.entities.AlignmentLayout attribute*), 274
dx_entities() (*in module ezdxf.addons.geo*), 546
dxfattribs() (*ezdxf.entities.DXFEntity method*), 282
dxfattribs() (*ezdxf.math.ConstructionEllipse method*), 443
DXFAttributeError (*class in ezdxf.lldxflldxf.const*), 393
DXFBinaryTag (*class in ezdxf.lldxflldxf.types*), 689
DXFBlockInUseError (*class in ezdxf.lldxflldxf.const*), 393
DXFClass (*class in ezdxf.entities*), 210
DXFEntity (*class in ezdxf.entities*), 281
DXFError (*class in ezdxf.lldxflldxf.const*), 393
DXFGraphic (*class in ezdxf.entities*), 284
DXFGroup (*class in ezdxf.entities.dxfgroups*), 279
DXFIndexError (*class in ezdxf.lldxflldxf.const*), 393
DXFInvalidLineType (*class in ezdxf.lldxflldxf.const*), 393
DXFKeyError (*class in ezdxf.lldxflldxf.const*), 393
DXFLayout (*class in ezdxf.entities*), 376
DXFOBJECT (*class in ezdxf.entities*), 377
dxfrstr() (*ezdxf.lldxflldxf.types.DXFBinaryTag method*), 689

dxfsstr() (*ezdxf.lldx.types.DXFTag method*), 689
dxfsstr() (*ezdxf.lldx.types.DXFVertex method*), 690
DXFStructureError (*class in ezdxf.lldx.const*), 393
DXFTableEntryError (*class in ezdxf.lldx.const*),
 393
DXFTag (*class in ezdxf.lldx.types*), 689
dxftag() (*in module ezdxf.lldx.types*), 688
dxftags() (*ezdxf.lldx.types.DXFVertex method*), 690
dxftype() (*ezdxf.entities.DXFEntity method*), 281
dxftype() (*ezdxf.lldx.extendedtags.ExtendedTags
 method*), 692
dxftype() (*ezdxf.lldx.tags.Tags method*), 690
DXFTypeError (*class in ezdxf.lldx.const*), 393
DXFUnclassifiedBlockError (*class
 in ezdxf.lldx.const*), 393
DXFValueError (*class in ezdxf.lldx.const*), 393
dxfversion (*ezdxf.document.Drawing attribute*), 199
DXFVersionError (*class in ezdxf.lldx.const*), 393
DXFVertex (*class in ezdxf.lldx.types*), 690

E

EDGE (*ezdxf.entities.BoundaryPathType attribute*), 308
edge_collapse_values (*ezdxf.entities.MeshData attribute*), 326
edge_to_polyline_paths() (*ezdxf.entities.BoundaryPaths method*), 307
EdgePath (*class in ezdxf.entities*), 308
edges (*ezdxf.entities.EdgePath attribute*), 309
edges (*ezdxf.entities.Mesh attribute*), 326
edges (*ezdxf.entities.MeshData attribute*), 326
EdgeType (*class in ezdxf.entities*), 310
edit_data() (*ezdxf.entities.dxfgroups.DXFGroup
 method*), 279
edit_data() (*ezdxf.entities.Mesh method*), 326
elements (*ezdxf.entities.ezdxf.entities.mline.MLineStyleElements
 attribute*), 325
elements (*ezdxf.entities.MLineStyle attribute*), 325
elevation (*ezdxf.entities.Hatch.dxf attribute*), 303
elevation (*ezdxf.entities.LWPolyline.dxf attribute*),
 320
elevation (*ezdxf.entities.MPolygon.dxf attribute*), 328
elevation (*ezdxf.entities.Polyline.dxf attribute*), 344
elevation (*ezdxf.entities.View.dxf attribute*), 238
elevation (*ezdxf.entities.Viewport.dxf attribute*), 371
Ellipse (*class in ezdxf.entities*), 300
ELLIPSE (*ezdxf.entities.EdgeType attribute*), 311
ellipse() (*in module ezdxf.render.forms*), 529
ellipse_approximation() (*ezdxf.math.BSpline
 static method*), 449
ellipse_edges_to_spline_edges() (*ezdxf.entities.BoundaryPaths method*), 307
ellipse_param_span() (*in module ezdxf.math*),
 409
EllipseEdge (*class in ezdxf.entities*), 311
elliptic_transformation() (*in module
 ezdxf.path*), 471
embed_mttext() (*ezdxf.entities.AttDef method*), 248
embed_mttext() (*ezdxf.entities.Attrib method*), 247
embedded_objects (*ezdxf.lldx.extendedtags.ExtendedTags
 attribute*), 692
Empty-Path, 462
encode() (*ezdxf.document.Drawing method*), 202
encode() (*in module ezdxf.tools.crypt*), 490
encode_base64() (*ezdxf.document.Drawing
 method*), 202
encode_raw_color() (*in module ezdxf.colors*), 402
encoding (*ezdxf.document.Drawing attribute*), 199
end (*ezdxf.entities.Line.dxf attribute*), 317
end (*ezdxf.entities.LineEdge attribute*), 311
end (*ezdxf.math.ConstructionEllipse attribute*), 443
end (*ezdxf.math.ConstructionLine attribute*), 436
end_angle (*ezdxf.entities.Arc.dxf attribute*), 289
end_angle (*ezdxf.entities.ArcDimension.dxf attribute*),
 300
end_angle (*ezdxf.entities.ArcEdge attribute*), 311
end_angle (*ezdxf.entities.EllipseEdge attribute*), 311
end_angle (*ezdxf.entities.MLineStyle.dxf attribute*),
 325
end_angle (*ezdxf.math.ConstructionArc attribute*),
 439
end_angle_rad (*ezdxf.math.ConstructionArc
 attribute*), 439
end_draft_angle (*ezdxf.entities.LoftedSurface.dxf
 attribute*), 359
end_draft_distance
 (*ezdxf.entities.RevolvedSurface.dxf attribute*),
 359
end_draft_magnitude
 (*ezdxf.entities.LoftedSurface.dxf attribute*),
 359
end_param (*ezdxf.entities.Ellipse.dxf attribute*), 300
end_point (*ezdxf.entities.Arc attribute*), 289
end_point (*ezdxf.entities.Ellipse attribute*), 301
end_point (*ezdxf.math.ConstructionArc attribute*),
 439
end_point (*ezdxf.math.ConstructionEllipse attribute*),
 443
end_style (*ezdxf.addons.acadctb.PlotStyle attribute*),
 577
end_tangent (*ezdxf.entities.Spline.dxf attribute*), 356
end_tangent (*ezdxf.entities.SplineEdge attribute*),
 312
end_width (*ezdxf.entities.Vertex.dxf attribute*), 348
EndBlk (*class in ezdxf.entities*), 242
Engineering (*ezdxf.enums.LengthUnits attribute*),
 401
entities (*ezdxf.document.Drawing attribute*), 200
entities_to_code() (*in module*

ezdxf.addons.dxf2code), 553
entity (ezdxf.disassemble.Primitive attribute), 476
entity() (ezdxf.entities.xdata.XDataUserDict class method), 486
entity() (ezdxf.entities.xdata.XDataUserList class method), 485
EntityDB (class in ezdxf.entitydb), 687
EntityQuery (class in ezdxf.query), 405
EntitySection (class in ezdxf.sections.entities), 213
EntitySpace (class in ezdxf.entitydb), 688
estimate_end_tangent_magnitude() (in module ezdxf.math), 417
estimate_mttext_content_extents() (in module ezdxf.tools.text), 497
estimate_mttext_extents() (in module ezdxf.tools.text), 497
estimate_mttext_extents() (in module ezdxf.tools.text_size), 499
estimate_tangents() (in module ezdxf.math), 416
euler_spiral() (in module ezdxf.render.forms), 529
EulerSpiral (class in ezdxf.math), 454
EulerSpiral (class in ezdxf.render), 526
EXACT (ezdxf.enums.MTextLineSpacing attribute), 399
example_func() (in module guide), 698
example_method() (guide.ExampleCls method), 699
ExampleCls (class in guide), 699
expand() (ezdxf.math.ConstructionBox method), 445
explode (ezdxf.entities.BlockRecord.dxf attribute), 240
explode() (ezdxf.addons.MTextExplode method), 567
explode() (ezdxf.entities.Dimension method), 295
explode() (ezdxf.entities.Insert method), 246
explode() (ezdxf.entities.Leader method), 317
explode() (ezdxf.entities.LWPolyline method), 322
explode() (ezdxf.entities.MLine method), 324
explode() (ezdxf.entities.MultiLeader method), 338
explode() (ezdxf.entities.Polyline method), 347
explore() (in module ezdxf.recover), 208
export() (ezdxf.addons.iterdxf.IterDXF method), 557
export_dwg() (in module ezdxf.addons.odafc), 565
export_dxf() (ezdxf.lldx.dxf.packedtags.VertexArray method), 695
extend() (ezdxf.entities.dxfgroups.DXFGGroup method), 279
extend() (ezdxf.entities.MLine method), 323
extend() (ezdxf.entities.XRecord method), 388
extend() (ezdxf.entitydb.EntitySpace method), 688
extend() (ezdxf.lldx.dxf.packedtags.VertexArray method), 695
extend() (ezdxf.math.BoundingBox method), 433
extend() (ezdxf.math.BoundingBox2d method), 434
extend() (ezdxf.math.Shape2d method), 447
extend() (ezdxf.query.EntityQuery method), 405
extend_multi_path() (ezdxf.path.Path method), 474
ExtendedTags (class in ezdxf.lldx.dxf.extendedtags), 692
ExtensionDict (class in ezdxf.entities.xdict), 390
extents() (in module ezdxf.bbox), 477
extents() (in module ezdxf.zoom), 510
external_paths() (ezdxf.entities.BoundaryPaths method), 306
extmax (ezdxf.math.BoundingBox attribute), 432
extmax (ezdxf.math.BoundingBox2d attribute), 434
extmin (ezdxf.math.BoundingBox attribute), 432
extmin (ezdxf.math.BoundingBox2d attribute), 433
extrude() (in module ezdxf.render.forms), 532
ExtrudedSurface (class in ezdxf.entities), 358
extrusion (ezdxf.entities.BlockData attribute), 342
extrusion (ezdxf.entities.DXFGraphic.dxf attribute), 286
extrusion (ezdxf.entities.Line.dxf attribute), 317
extrusion (ezdxf.entities.MLine.dxf attribute), 323
extrusion (ezdxf.entities.MTextData attribute), 341
extrusion (ezdxf.entities.Underlay.dxf attribute), 366
extrusion (ezdxf.math.ConstructionEllipse attribute), 442
ezdxf.addons (module), 540
ezdxf.addons.acadctb (module), 574
ezdxf.addons.drawing (module), 540
ezdxf.addons.drawing.backend.Backend (class in ezdxf.addons.drawing), 544
ezdxf.addons.drawing.frontend.Frontend (class in ezdxf.addons.drawing), 544
ezdxf.addons.drawing.matplotlib.MatplotlibBackend (class in ezdxf.addons.drawing), 541
ezdxf.addons.drawing.properties.LayerProperties (class in ezdxf.addons.drawing), 544
ezdxf.addons.drawing.properties.Properties (class in ezdxf.addons.drawing), 544
ezdxf.addons.drawing.properties.RenderContext (class in ezdxf.addons.drawing), 544
ezdxf.addons.drawing.pyqt.QtBackend (class in ezdxf.addons.drawing), 541
ezdxf.addons.dxf2code (module), 553
ezdxf.addons.geo (module), 545
ezdxf.addons.importer (module), 549
ezdxf.addons.iterdxf (module), 555
ezdxf.addons.odafc (module), 564
ezdxf.addons.pycsg (module), 568
ezdxf.addons.r12writer (module), 558
ezdxf.addons.text2path (module), 566
ezdxf.bbox (module), 476
ezdxf.blkrefs (module), 391
ezdxf.colors (module), 402
ezdxf.comments (module), 510
ezdxf.disassemble (module), 474
ezdxf.document (module), 199
ezdxf.document.MetaData (built-in class), 198
ezdxf.entities (module), 280

ezdxf.entities.appdata (*module*), 697
ezdxf.entities.dxfgroups (*module*), 279
ezdxf.entities.mline.MLineStyleElement
 (*class in ezdxf.entities*), 325
ezdxf.entities.mline.MLineStyleElements
 (*class in ezdxf.entities*), 325
ezdxf.entities.xdata (*module*), 695
ezdxf.entities.xdict (*module*), 389
ezdxf.entitydb (*module*), 687
ezdxf.enums (*module*), 397
ezdxf.gfxattribs (*module*), 490
ezdxf.layouts (*module*), 249
ezdxf.lldxf.const (*module*), 392
ezdxf.lldxf.const.MTextParagraphAlignmen
 (*class in ezdxf.tools.text*), 495
ezdxf.lldxf.extendedtags (*module*), 692
ezdxf.lldxf.packedtags (*module*), 693
ezdxf.lldxf.tags (*module*), 690
ezdxf.lldxf.types (*module*), 688
ezdxf.math (*module*), 407
ezdxf.options (*module*), 503
ezdxf.path (*module*), 461
ezdxf.query (*module*), 404
ezdxf.recover (*module*), 204
ezdxf.render (*module*), 523
ezdxf.render.arrows (*module*), 540
ezdxf.render.forms (*module*), 527
ezdxf.render.point (*module*), 539
ezdxf.render.trace (*module*), 537
ezdxf.reorder (*module*), 510
ezdxf.sections.blocks (*module*), 212
ezdxf.sections.classes (*module*), 210
ezdxf.sections.entities (*module*), 213
ezdxf.sections.header (*module*), 209
ezdxf.sections.objects (*module*), 213
ezdxf.sections.table (*module*), 216
ezdxf.sections.tables (*module*), 211
ezdxf.tools.fonts (*module*), 500
ezdxf.tools.text (*module*), 492
ezdxf.tools.text_size (*module*), 499
ezdxf.tools.text_size.MTextSize
 (*class in ezdxf.tools.text_size*), 499
ezdxf.tools.text_size.TextSize
 (*class in ezdxf.tools.text_size*), 499
ezdxf.units (*module*), 35
ezdxf.upright (*module*), 480
ezdxf.urecord (*module*), 486
ezdxf.zoom (*module*), 509
ezdxf_metadata () (*ezdxf.document.Drawing*
 method), 204

F

Face3d (*class in ezdxf.entities*), 287
faces (*ezdxf.entities.GeoData* *attribute*), 378

faces (*ezdxf.entities.Mesh* *attribute*), 326
faces (*ezdxf.entities.MeshData* *attribute*), 326
faces (*ezdxf.render.MeshBuilder* *attribute*), 533
faces () (*ezdxf.entities.Polyface* *method*), 350
faces () (*ezdxf.render.trace.CurvedTrace* *method*), 539
faces () (*ezdxf.render.trace.LinearTrace* *method*), 538
faces () (*ezdxf.render.trace.TraceBuilder* *method*), 537
faces_as_vertices () (*ezdxf.render.MeshBuilder*
 method), 533
faces_wcs () (*ezdxf.render.trace.TraceBuilder*
 method), 537
fade (*ezdxf.entities.Image*.*dxftype* *attribute*), 314
fade (*ezdxf.entities.Underlay*.*dxftype* *attribute*), 367
family (*ezdxf.tools.fonts.FontFace* *attribute*), 501
fast_plain_mtext () (*in module ezdxf.tools.text*),
 497
fast_zoom (*ezdxf.entities.VPort*.*dxftype* *attribute*), 236
Feet (*ezdxf.enums.InsertUnits* *attribute*), 400
field_length (*ezdxf.entities.AttDef*.*dxftype* *attribute*),
 248
filename (*ezdxf.document.Drawing* *attribute*), 200
filename (*ezdxf.entities.ImageDef*.*dxftype* *attribute*), 380
filename (*ezdxf.entities.UnderlayDefinition*.*dxftype* *attribute*), 386
fill_color (*ezdxf.entities.MLineStyle*.*dxftype* *attribute*),
 325
fill_params (*ezdxf.entities.MLineVertex* *attribute*),
 324
fill_style (*ezdxf.addons.acadctb.PlotStyle* *attribute*),
 577
filter () (*ezdxf.addons.geo.GeoProxy* *method*), 549
filter () (*ezdxf.lldxf.tags.Tags* *method*), 691
filter_invalid_xdata_group_codes (*in module*
 ezdxf.options), 508
finalize () (*ezdxf.addons.importer.Importer* *method*),
 551
finalize () (*ezdxf.addons.MTextExplode* *method*),
 567
find_all () (*ezdxf.lldxf.tags.Tags* *method*), 691
find_shx () (*ezdxf.sections.table.TextstyleTable*
 method), 218
first (*ezdxf.query.EntityQuery* *attribute*), 405
first_segment_angle_constraint
 (*ezdxf.entities.MLeaderStyle*.*dxftype* *attribute*),
 382
FIT (*ezdxf.enums.TextEntityAlignment* *attribute*), 398
fit_length () (*ezdxf.entities.Text* *method*), 364
fit_paths_into_box () (*in module ezdxf.path*), 469
fit_point_count () (*ezdxf.entities.Spline* *method*),
 356
fit_points (*ezdxf.entities.Spline* *attribute*), 356
fit_points (*ezdxf.entities.SplineEdge* *attribute*), 312
fit_points_to_cad_cv () (*in module ezdxf.math*),
 417

fit_points_to_cubic_bezier() (in module `ezdxf.math`), 418
 fit_tolerance (`ezdxf.entities.Spline.dxf` attribute), 355
 flag (`guide.ExampleCls` attribute), 699
 flags (`ezdxf.entities.AppID.dxf` attribute), 238
 flags (`ezdxf.entities.Block.dxf` attribute), 241
 flags (`ezdxf.entities.Body.dxf` attribute), 290
 flags (`ezdxf.entities.DimStyle.dxf` attribute), 227
 flags (`ezdxf.entities.DXFClass.dxf` attribute), 210
 flags (`ezdxf.entities.Image.dxf` attribute), 314
 flags (`ezdxf.entities.Layer.dxf` attribute), 221
 flags (`ezdxf.entities.LWPolyline.dxf` attribute), 320
 flags (`ezdxf.entities.MLine.dxf` attribute), 323
 flags (`ezdxf.entities.MLineStyle.dxf` attribute), 325
 flags (`ezdxf.entities.Polyline.dxf` attribute), 344
 flags (`ezdxf.entities.Spline.dxf` attribute), 355
 flags (`ezdxf.entities.Textstyle.dxf` attribute), 225
 flags (`ezdxf.entities.UCSTableEntry.dxf` attribute), 239
 flags (`ezdxf.entities.Underlay.dxf` attribute), 366
 flags (`ezdxf.entities.Vertex.dxf` attribute), 348
 flags (`ezdxf.entities.View.dxf` attribute), 236
 flags (`ezdxf.entities.Viewport.dxf` attribute), 369
 flags (`ezdxf.entities.VPort.dxf` attribute), 235
 flatten_subclasses()
 (`ezdxf.lldxf.extendedtags.ExtendedTags`
 method), 692
 flattening() (`ezdxf.entities.Arc` method), 289
 flattening() (`ezdxf.entities.Circle` method), 291
 flattening() (`ezdxf.entities.Ellipse` method), 301
 flattening() (`ezdxf.entities.Spline` method), 356
 flattening() (`ezdxf.math.Bezier` method), 450
 flattening() (`ezdxf.math.Bezier3P` method), 452
 flattening() (`ezdxf.math.Bezier4P` method), 452
 flattening() (`ezdxf.math.BSpline` method), 448
 flattening() (`ezdxf.math.ConstructionCircle`
 method), 437
 flattening() (`ezdxf.math.ConstructionEllipse`
 method), 443
 flattening() (`ezdxf.path.Path` method), 474
 float2transparency() (in module `ezdxf.colors`),
 402
 flow_direction (`ezdxf.entities.MText.dxf` attribute),
 331
 flow_direction (`ezdxf.entities.MTextData` at-
 tribute), 341
 font (`ezdxf.entities.Textstyle.dxf` attribute), 225
 font() (`ezdxf.tools.text.MTextEditor` method), 493
 font_cache_directory (in module `ezdxf.options`),
 507
 font_measurements() (`ezdxf.tools.text.TextLine`
 method), 496
 font_name() (`ezdxf.entities.Text` method), 363
 FontFace (class in `ezdxf.tools.fonts`), 501
 FontMeasurements (class in `ezdxf.tools.fonts`), 502
 format() (`ezdxf.entities.Vertex` method), 348
 Fractional (`ezdxf.enums.LengthUnits` attribute), 401
 freeze() (`ezdxf.entities.Layer` method), 222
 freeze() (`ezdxf.math.Matrix` method), 459
 freeze_matrix() (in module `ezdxf.math`), 457
 from_2p_angle() (`ezdxf.math.ConstructionArc` class
 method), 440
 from_2p_radius() (`ezdxf.math.ConstructionArc`
 class method), 440
 from_3p() (`ezdxf.math.ConstructionArc` class
 method), 440
 from_3p() (`ezdxf.math.ConstructionCircle` static
 method), 437
 from_3p() (`ezdxf.math.Plane` class method), 432
 from_angle() (`ezdxf.math.Vec3` class method), 429
 from_arc() (`ezdxf.entities.Ellipse` class method), 301
 from_arc() (`ezdxf.entities.Spline` class method), 357
 from_arc() (`ezdxf.math.BSpline` static method), 449
 from_arc() (`ezdxf.math.ConstructionEllipse` class
 method), 443
 from_arc() (`ezdxf.render.trace.CurvedTrace` class
 method), 539
 from_builder() (`ezdxf.render.MeshBuilder` class
 method), 535
 from_deg_angle() (`ezdxf.math.Vec3` class method),
 429
 from_dxf_entities()
 (`ezdxf.addons.geo.GeoProxy` class method),
 547
 from_ellipse() (`ezdxf.math.BSpline` static method),
 449
 from_entity() (`ezdxf.gfxattribs.GfxAttribs` class
 method), 492
 from_file() (in module `ezdxf.comments`), 510
 from_fit_points() (`ezdxf.math.BSpline` static
 method), 449
 from_hatch() (in module `ezdxf.path`), 462
 from_matplotlib_path() (in module `ezdxf.path`),
 463
 from_mesh() (`ezdxf.render.MeshBuilder` class
 method), 535
 from_points() (`ezdxf.math.ConstructionBox` class
 method), 445
 from_polyface() (`ezdxf.render.MeshBuilder` class
 method), 535
 from_polyline() (`ezdxf.render.trace.TraceBuilder`
 class method), 538
 from_profiles_linear() (in module
 `ezdxf.render.forms`), 532
 from_profiles_spline() (in module
 `ezdxf.render.forms`), 532
 from_qpainter_path() (in module `ezdxf.path`), 463
 from_spline() (`ezdxf.render.trace.CurvedTrace`

class method), 539
from_stream() (*in module eздxf.comments*), 510
from_tags() (*ezdxf.lldxf.packedtags.TagList class method*), 694
from_tags() (*ezdxf.lldxf.packedtags.VertexArray class method*), 695
from_text() (*ezdxf.lldxf.extendedtags.ExtendedTags class method*), 693
from_text() (*ezdxf.lldxf.tags.Tags class method*), 690
from_vector() (*ezdxf.math.Plane class method*), 432
from_vertices() (*in module eздxf.path*), 463
from_wcs() (*ezdxf.math.OCS method*), 421
from_wcs() (*ezdxf.math.UCS method*), 422
from_x_axis_and_point_in_xy()
 (*ezdxf.math.UCS static method*), 423
from_x_axis_and_point_in_xz()
 (*ezdxf.math.UCS static method*), 423
from_y_axis_and_point_in_xy()
 (*ezdxf.math.UCS static method*), 424
from_y_axis_and_point_in_yz()
 (*ezdxf.math.UCS static method*), 424
from_z_axis_and_point_in_xz()
 (*ezdxf.math.UCS static method*), 424
from_z_axis_and_point_in_yz()
 (*ezdxf.math.UCS static method*), 424
front_clip_plane_z_value
 (*ezdxf.entities.Viewport.dxf attribute*), 369
front_clipping (*ezdxf.entities.View.dxf attribute*),
 237
front_clipping (*ezdxf.entities.VPort.dxf attribute*),
 235
frozen_layers (*ezdxf.entities.Viewport attribute*),
 372

G

gauss_jordan_inverse() (*in module eздxf.math*),
 455
gauss_jordan_solver() (*in module eздxf.math*),
 455
gauss_matrix_solver() (*in module eздxf.math*),
 455
gauss_vector_solver() (*in module eздxf.math*),
 455
gear() (*in module eздxf.path*), 472
gear() (*in module eздxf.render.forms*), 530
generate() (*ezdxf.math.Vec3 class method*), 429
generate_geometry() (*ezdxf.entities.MLine method*), 324
generation_flags (*ezdxf.entities.Textstyle.dxf attribute*), 225
geo_rss_tag (*ezdxf.entities.GeoData.dxf attribute*),
 378
GeoData (*class in eздxf.entities*), 377
geometry (*ezdxf.entities.Dimension.dxf attribute*), 292
GeoProxy (*class in eздxf.addons.geo*), 547
geotype (*ezdxf.addons.geo.GeoProxy attribute*), 547
get() (*ezdxf.document.eздxf.document.MetaData.MetaData method*), 199
get() (*ezdxf.entities.appdata.AppData method*), 697
get() (*ezdxf.entities.appdata.Reactors method*), 698
get() (*ezdxf.entities.Dictionary method*), 375
get() (*ezdxf.entities.DictionaryWithDefault method*),
 376
get() (*ezdxf.entities.DimStyleOverride method*), 296
get() (*ezdxf.entities.dxfgroups.GroupCollection method*), 280
get() (*ezdxf.entities.xdata.XData method*), 696
get() (*ezdxf.entities.xdict.ExtensionDict method*), 390
get() (*ezdxf.entitydb.EntityDB method*), 687
get() (*ezdxf.layouts.Layouts method*), 249
get() (*ezdxf.sections.blocks.BlocksSection method*),
 212
get() (*ezdxf.sections.classes.ClassesSection method*),
 210
get() (*ezdxf.sections.header.CustomVars method*), 209
get() (*ezdxf.sections.header.HeaderSection method*),
 209
get() (*ezdxf.sections.table.Table method*), 216
get() (*in module eздxf.options*), 505
get_align() (*ezdxf.entities.Text method*), 363
get_align_enum() (*ezdxf.entities.Text method*), 363
get_app_data() (*ezdxf.entities.DXFEntity method*),
 282
get_app_data() (*ezdxf.lldxf.extendedtags.ExtendedTags method*), 693
get_app_data_content()
 (*ezdxf.lldxf.extendedtags.ExtendedTags method*), 693
get_arrow_names()
 (*ezdxf.entities.DimStyleOverride method*),
 296
get_attdef() (*ezdxf.layouts.BlockLayout method*),
 278
get_attdef_text() (*ezdxf.layouts.BlockLayout method*), 278
get_attrib() (*ezdxf.entities.Insert method*), 244
get_attrib_text() (*ezdxf.entities.Insert method*),
 244
get_bool() (*in module eздxf.options*), 505
get_col() (*ezdxf.math.Matrix44 method*), 425
get_color() (*ezdxf.entities.Layer method*), 223
get_config() (*ezdxf.sections.table.ViewportTable method*), 220
get_crs() (*ezdxf.entities.GeoData method*), 378
get_crs_transformation()
 (*ezdxf.entities.GeoData method*), 379
get_dim_style() (*ezdxf.entities.Dimension method*), 294

get_dxattrib() (*ezdxf.entities.DXFEntity method*), 281
 get_entity_font_face() (*in module ezdxf.tools.fonts*), 502
 get_extended_font_data() (*ezdxf.entities.Textstyle method*), 226
 get_extension_dict() (*ezdxf.entities.DXFEntity method*), 282
 get_extension_dict() (*ezdxf.layouts.BaseLayout method*), 251
 get_first_tag() (*ezdxf.lldx.tags.Tags method*), 690
 get_first_value() (*ezdxf.lldx.tags.Tags method*), 691
 get_flag_state() (*ezdxf.entities.DXFEntity method*), 282
 get_float() (*in module ezdxf.options*), 505
 get_font_face() (*in module ezdxf.tools.fonts*), 502
 get_font_measurements() (*in module ezdxf.tools.fonts*), 502
 get_geodata() (*ezdxf.layouts.Modelspace method*), 277
 get_geometry_block() (*ezdxf.entities.Dimension method*), 294
 get_handle() (*ezdxf.lldx.extendedtags.ExtendedTags method*), 692
 get_handle() (*ezdxf.lldx.tags.Tags method*), 690
 get_hyperlink() (*ezdxf.entities.DXFGraphic method*), 285
 get_int() (*in module ezdxf.options*), 505
 get_layout() (*ezdxf.entities.DXFGraphic method*), 284
 get_layout_for_entity() (*ezdxf.layouts.Layouts method*), 250
 get_lineweight() (*ezdxf.addons.acadctb.ColorDependentPlotStyles method*), 575
 get_lineweight() (*ezdxf.addons.acadctb.NamedPlotStyles method*), 576
 get_lineweight_index() (*ezdxf.addons.acadctb.ColorDependentPlotStyles method*), 575
 get_lineweight_index() (*ezdxf.addons.acadctb.NamedPlotStyles method*), 576
 get_locations() (*ezdxf.entities.MLine method*), 323
 get_measurement() (*ezdxf.entities.Dimension method*), 295
 get_mesh_vertex() (*ezdxf.entities.Polymesh method*), 349
 get_mesh_vertex_cache() (*ezdxf.entities.Polymesh method*), 349
 get_mode() (*ezdxf.entities.Polyline method*), 346
 get_paper_limits() (*ezdxf.layouts.Paperspace method*), 278
 get_placement() (*ezdxf.entities.Text method*), 363
 get_points() (*ezdxf.entities.LWPolyline method*), 321
 get_pos() (*ezdxf.entities.Text method*), 363
 get_reactors() (*ezdxf.entities.DXFEntity method*), 283
 get_redraw_order() (*ezdxf.layouts.Layout method*), 276
 get_required_dict() (*ezdxf.entities.Dictionary method*), 375
 get_rotation() (*ezdxf.entities.MText method*), 332
 get_row() (*ezdxf.math.Matrix44 method*), 425
 get_shx() (*ezdxf.sections.table.TextstyleTable method*), 218
 get_subclass() (*ezdxf.lldx.extendedtags.ExtendedTags method*), 693
 get_table_lineweight() (*ezdxf.addons.acadctb.ColorDependentPlotStyles method*), 575
 get_table_lineweight() (*ezdxf.addons.acadctb.NamedPlotStyles method*), 576
 get_text_direction() (*ezdxf.entities.MText method*), 332
 get_underlay_def() (*ezdxf.entities.Underlay method*), 367
 get_xdata() (*ezdxf.entities.DXFEntity method*), 283
 get_xdata() (*ezdxf.lldx.extendedtags.ExtendedTags method*), 693
 get_xdata_list() (*ezdxf.entities.DXFEntity method*), 283
 get_xlist() (*ezdxf.entities.XData method*), 696
 GfxAttribs (*class in ezdxf.gfxattribs*), 491
 GlobalParameters (*ezdxf.enums.InsertUnits attribute*), 400
 global_bspline_interpolation() (*in module ezdxf.math*), 418
 globe_to_map() (*ezdxf.addons.geo.GeoProxy method*), 548
 Grad (*ezdxf.enums.AngularUnits attribute*), 401
 Gradient (*class in ezdxf.entities*), 313
 gradient (*ezdxf.entities.Hatch attribute*), 303
 gradient (*ezdxf.entities.MPolygon attribute*), 328
 graphic_properties() (*ezdxf.entities.DXFGraphic method*), 285
 grayscale (*ezdxf.addons.acadctb.PlotStyle attribute*), 578
 GREEN (*ezdxf.enums.AC1 attribute*), 402
 grid() (*ezdxf.entities.Insert method*), 244
 grid_frequency (*ezdxf.entities.Viewport.dxf attribute*), 371
 grid_on (*ezdxf.entities.VPort.dxf attribute*), 236

grid_spacing (*ezdxf.entities.Viewport.dxf attribute*), 369
grid_spacing (*ezdxf.entities.VPort.dxf attribute*), 235
group() (*ezdxf.tools.text.MTextEditor method*), 493
group_tags() (*in module ezdxf.lldx.tags*), 692
groupby() (*ezdxf.document.Drawing method*), 202
groupby() (*ezdxf.layouts.BaseLayout method*), 251
groupby() (*ezdxf.query.EntityQuery method*), 406
groupby() (*in module ezdxf.groupby*), 406
GroupCollection (*class in ezdxf.entities.dxfgroups*), 280
groups (*ezdxf.document.Drawing attribute*), 200
groups() (*ezdxf.entities.dxfgroups.GroupCollection method*), 280
guid() (*in module ezdxf.tools*), 489
guide (*module*), 698
gutter_width (*ezdxf.tools.text_size.ezdxf.tools.text_size.MTextSize*.*kline* (*ezdxf.entities.Leader.dxf attribute*), 499

H

halign (*ezdxf.entities.Text.dxf attribute*), 361
handle (*ezdxf.entities.ArrowHeadData attribute*), 340
handle (*ezdxf.entities.AttribData attribute*), 340
handle (*ezdxf.entities.Block.dxf attribute*), 241
handle (*ezdxf.entities.DXFEntity.dxf attribute*), 281
handle (*ezdxf.entities.EndBlk.dxf attribute*), 242
handle (*ezdxf.entities.Layer.dxf attribute*), 221
handle (*ezdxf.entities.Textstyle.dxf attribute*), 225
handles() (*ezdxf.entities.dxfgroups.DXFGGroup method*), 279
hard_owned (*ezdxf.entities.Dictionary.dxf attribute*), 374
has_app_data() (*ezdxf.entities.DXFEntity method*), 282
has_app_data() (*ezdxf.lldx.extendedtags.ExtendedTag method*), 693
has_arc (*ezdxf.entities.Polyline attribute*), 346
has_arrowhead (*ezdxf.entities.Leader.dxf attribute*), 316
has_attdef() (*ezdxf.layouts.BlockLayout method*), 278
has_attrib() (*ezdxf.entities.Insert method*), 244
has_bg_fill (*ezdxf.entities.MTextData attribute*), 342
has_binary_data (*ezdxf.entities.Body attribute*), 290
has_block_rotation
 (*ezdxf.entities.MLeaderStyle.dxf attribute*), 382
has_block_scaling
 (*ezdxf.entities.MLeaderStyle.dxf attribute*), 382
has_clockwise_orientation() (*ezdxf.path.Path method*), 473
has_dogleg (*ezdxf.entities.MLeaderStyle.dxf attribute*), 382
has_dogleg (*ezdxf.entities.MultiLeader.dxf attribute*), 336
has_dogleg_vector (*ezdxf.entities.LeaderData attribute*), 340
has_dxfs_attrib() (*ezdxf.entities.DXFEntity method*), 281
has_dxfs_unicode() (*in module ezdxf*), 489
has_embedded_objects() (*ezdxf.lldx.tags.Tags method*), 690
has_entry() (*ezdxf.sections.table.Table method*), 216
has_extension_dict (*ezdxf.entities.DXFEntity attribute*), 282
has_handle() (*ezdxf.blkrefs.BlockDefinitionIndex method*), 392
has_handle() (*ezdxf.entitydb.EntitySpace method*), 688
has_hypeline () (*ezdxf.entities.DXFGGraphic method*), 285
has_intersection() (*ezdxf.math.ConstructionLine method*), 436
has_landing (*ezdxf.entities.MLeaderStyle.dxf attribute*), 382
has_landing (*ezdxf.entities.MultiLeader.dxf attribute*), 336
has_last_leader_line (*ezdxf.entities.LeaderData attribute*), 340
has_leader (*ezdxf.entities.ArcDimension.dxf attribute*), 300
has_matrix_2d_stretching() (*in module ezdxf.math*), 409
has_matrix_3d_stretching() (*in module ezdxf.math*), 409
has_name() (*ezdxf.blkrefs.BlockDefinitionIndex method*), 392
has_none_planar_faces() (*ezdxf.render.MeshBuilder method*), 534
has_reactors() (*ezdxf.entities.DXFEntity method*), 283
has_scaling (*ezdxf.entities.Insert attribute*), 243
has_tag() (*ezdxf.lldx.tags.Tags method*), 690
has_tag() (*ezdxf.sections.header.CustomVars method*), 209
has_text_frame (*ezdxf.entities.MultiLeader.dxf attribute*), 336
has_uniform_scaling (*ezdxf.entities.Insert attribute*), 243
has_width (*ezdxf.entities.Polyline attribute*), 346
has_xdata() (*ezdxf.entities.DXFEntity method*), 282
has_xdata() (*ezdxf.lldx.extendedtags.ExtendedTags method*), 693
has_xdata_list() (*ezdxf.entities.DXFEntity method*), 283

has_xlist() (*ezdxf.entities.XData method*), 696
Hatch (*class in ezdxf.entities*), 302
hatch_policy (*ezdxf.addons.drawing.config.Configuration attribute*), 542
hatch_style (*ezdxf.entities.Hatch.dxf attribute*), 303
HatchPolicy (*class in ezdxf.addons.drawing.config*), 543
have_bezier_curves_g1_continuity() (*in module ezdxf.math*), 421
have_close_control_vertices() (*in module ezdxf.path*), 471
header (*ezdxf.document.Drawing attribute*), 200
HeaderSection (*class in ezdxf.sections.header*), 209
Hectometers (*ezdxf.enums.InsertUnits attribute*), 400
height (*ezdxf.entities.Text.dxf attribute*), 361
height (*ezdxf.entities.Textstyle.dxf attribute*), 225
height (*ezdxf.entities.View.dxf attribute*), 236
height (*ezdxf.entities.Viewport.dxf attribute*), 368
height (*ezdxf.entities.VPort.dxf attribute*), 235
height (*ezdxf.math.ConstructionBox attribute*), 444
height() (*ezdxf.tools.text.MTextEditor method*), 493
history_handle (*ezdxf.entities.Solid3d.dxf attribute*), 288
hits (*ezdxf.bbox.Cache attribute*), 479
hookline_direction (*ezdxf.entities.Leader.dxf attribute*), 316
horizontal_direction
 (*ezdxf.entities.Dimension.dxf attribute*), 294
horizontal_direction (*ezdxf.entities.Leader.dxf attribute*), 316
horizontal_unit_scale
 (*ezdxf.entities.GeoData.dxf attribute*), 378
horizontal_units (*ezdxf.entities.GeoData.dxf attribute*), 378

|

id (*ezdxf.entities.Viewport.dxf attribute*), 369
identity() (*ezdxf.math.Matrix class method*), 458
IGNORE (*ezdxf.addons.drawing.config.HatchPolicy attribute*), 543
IGNORE (*ezdxf.addons.drawing.config.ProxyGraphicPolicy attribute*), 543
Image (*class in ezdxf.entities*), 314
image_def (*ezdxf.entities.Image attribute*), 315
image_def_handle (*ezdxf.entities.Image.dxf attribute*), 314
image_handle (*ezdxf.entities.ImageDefReactor.dxf attribute*), 380
image_size (*ezdxf.entities.Image.dxf attribute*), 314
image_size (*ezdxf.entities.ImageDef.dxf attribute*), 380
ImageDef (*class in ezdxf.entities*), 380
ImageDefReactor (*class in ezdxf.entities*), 380

Imperial (*ezdxf.enums.Measurement attribute*), 400
import_block() (*ezdxf.addons.importer.Importer method*), 551
import_blocks() (*ezdxf.addons.importer.Importer method*), 551
import_entities()
 (*ezdxf.addons.importer.Importer method*), 552
import_entity() (*ezdxf.addons.importer.Importer method*), 552
import_modelspace()
 (*ezdxf.addons.importer.Importer method*), 552
import_paperspace_layout()
 (*ezdxf.addons.importer.Importer method*), 552
import_paperspace_layouts()
 (*ezdxf.addons.importer.Importer method*), 552
import_shape_files()
 (*ezdxf.addons.importer.Importer method*), 552
import_str() (*ezdxf.addons.dxf2code.Code method*), 555
import_table() (*ezdxf.addons.importer.Importer method*), 552
import_tables() (*ezdxf.addons.importer.Importer method*), 553
Importer (*class in ezdxf.addons.importer*), 550
imports (*ezdxf.addons.dxf2code.Code attribute*), 554
Inches (*ezdxf.enums.InsertUnits attribute*), 400
incircle_radius (*ezdxf.math.ConstructionBox attribute*), 445
index (*ezdxf.addons.acadctb.PlotStyle attribute*), 577
index (*ezdxf.entities.ArrowHeadData attribute*), 340
index (*ezdxf.entities.AttribData attribute*), 341
index (*ezdxf.entities.LeaderData attribute*), 340
index (*ezdxf.entities.LeaderLine attribute*), 340
index (*ezdxf.math.BandedMatrixLU attribute*), 460
index_at() (*ezdxf.math.ConstructionPolyline method*), 446
infinite_line_length
 (*ezdxf.addons.drawing.config.Configuration attribute*), 542
Insert (*class in ezdxf.entities*), 243
insert (*ezdxf.entities.BlockData attribute*), 342
insert (*ezdxf.entities.Dimension.dxf attribute*), 293
insert (*ezdxf.entities.Image.dxf attribute*), 314
insert (*ezdxf.entities.Insert.dxf attribute*), 243
insert (*ezdxf.entities.MText.dxf attribute*), 330
insert (*ezdxf.entities.MTextData attribute*), 341
insert (*ezdxf.entities.Shape.dxf attribute*), 352
insert (*ezdxf.entities.Text.dxf attribute*), 361
insert (*ezdxf.entities.Underlay.dxf attribute*), 366

```
insert() (ezdxf.entities.LWPolyline method), 321
insert() (ezdxf.lldxf.packedtags.VertexArray method),
    695
insert_knot() (ezdxf.math.BSpline method), 449
insert_vertices() (ezdxf.entities.Polyline
    method), 347
InsertUnits (class in ezdxf.enums), 400
inside() (ezdxf.math.BoundingBox method), 433
inside() (ezdxf.math.BoundingBox2d method), 434
inside() (ezdxf.math.ConstructionCircle method),
    437
inside_bounding_box()
    (ezdxf.math.ConstructionLine method), 436
instance_count (ezdxf.entities.DXFClass.dxf at-
    tribute), 211
int2rgb() (in module ezdxf.colors), 402
intensity (ezdxf.entities.Sun.dxf attribute), 385
intersect() (ezdxf.addons.pyCSG.CSG method), 573
intersect() (ezdxf.math.BoundingBox method), 433
intersect() (ezdxf.math.BoundingBox2d method),
    434
intersect() (ezdxf.math.ConstructionBox method),
    445
intersect() (ezdxf.math.ConstructionLine method),
    436
intersect() (ezdxf.math.ConstructionRay method),
    435
intersect_arc() (ezdxf.math.ConstructionArc
    method), 442
intersect_circle() (ezdxf.math.ConstructionArc
    method), 441
intersect_circle()
    (ezdxf.math.ConstructionCircle method), 438
intersect_line() (ezdxf.math.ConstructionArc
    method), 441
intersect_line() (ezdxf.math.ConstructionCircle
    method), 438
intersect_polylines_2d() (in module
    ezdxf.math), 412
intersect_polylines_3d() (in module
    ezdxf.math), 416
intersect_ray() (ezdxf.math.ConstructionArc
    method), 441
intersect_ray() (ezdxf.math.ConstructionCircle
    method), 437
intersection_line_line_2d() (in module
    ezdxf.math), 411
intersection_line_line_3d() (in module
    ezdxf.math), 416
intersection_ray_ray_3d() (in module
    ezdxf.math), 416
invalidate() (ezdxf.bbox.Cache method), 479
inverse() (ezdxf.addons.pyCSG.CSG method), 574
inverse() (ezdxf.math.LUDecomposition method),
    460
inverse() (ezdxf.math.Matrix method), 459
inverse() (ezdxf.math.Matrix44 method), 427
invisible (ezdxf.entities.DXFGraphic.dxf attribute),
    286
invisible_edge (ezdxf.entities.Face3d.dxf attribute),
    287
is_2d_polyline (ezdxf.entities.Polyline attribute),
    345
is_2d_polyline_vertex (ezdxf.entities.Vertex at-
    tribute), 348
is_3d_polyline (ezdxf.entities.Polyline attribute),
    345
is_3d_polyline_vertex (ezdxf.entities.Vertex at-
    tribute), 348
is_active_paperspace (ezdxf.layouts.BaseLayout
    attribute), 251
is_alive (ezdxf.layouts.BaseLayout attribute), 250
is_an_entity (ezdxf.entities.DXFClass.dxf at-
    tribute), 211
is_annotation (ezdxf.entities.MLeaderStyle.dxf at-
    tribute), 382
is_annotation (ezdxf.entities.MultiLeader.dxf
    attribute), 336
is_anonymous (ezdxf.entities.Block attribute), 242
is_any_corner_inside()
    (ezdxf.math.ConstructionBox method), 445
is_any_layout (ezdxf.layouts.BaseLayout attribute),
    251
is_any_paperspace (ezdxf.layouts.BaseLayout at-
    tribute), 251
is_block_layout (ezdxf.layouts.BaseLayout at-
    tribute), 251
is_cartesian (ezdxf.math.UCS attribute), 422
is_closed (ezdxf.entities.Polyline attribute), 346
is_closed (ezdxf.entities.PolylinePath attribute), 308
is_coplanar_plane() (ezdxf.math.Plane method),
    432
is_coplanar_vertex() (ezdxf.math.Plane
    method), 432
is_face_record (ezdxf.entities.Vertex attribute), 348
is_frozen() (ezdxf.entities.Layer method), 222
is_hard_owner (ezdxf.entities.Dictionary attribute),
    374
is_horizontal (ezdxf.math.ConstructionLine at-
    tribute), 436
is_horizontal (ezdxf.math.ConstructionRay at-
    tribute), 435
is_inside() (ezdxf.math.ConstructionBox method),
    445
is_layout_block (ezdxf.entities.Block attribute),
    242
is_locked() (ezdxf.entities.Layer method), 222
```

i
 is_m_closed (*ezdxf.entities.Polyline attribute*), 346
 is_modelspace (*ezdxf.layouts.BaseLayout attribute*), 251
 is_n_closed (*ezdxf.entities.Polyline attribute*), 346
 is_null (*ezdxf.math.Vec3 attribute*), 428
 is_off () (*ezdxf.entities.Layer method*), 222
 is_on () (*ezdxf.entities.Layer method*), 222
 is_overlapping () (*ezdxf.math.ConstructionBox method*), 445
 is_parallel () (*ezdxf.math.ConstructionRay method*), 435
 is_parallel () (*ezdxf.math.Vec3 method*), 430
 is_partial (*ezdxf.entities.ArcDimension.dxf attribute*), 300
 is_planar_face () (*in module ezdxf.math*), 415
 is_point_in_polygon_2d () (*in module ezdxf.math*), 411
 is_point_left_of_line () (*ezdxf.math.ConstructionLine method*), 436
 is_point_left_of_line () (*in module ezdxf.math*), 411
 is_point_on_line_2d () (*in module ezdxf.math*), 411
 is_poly_face_mesh (*ezdxf.entities.Polyline attribute*), 346
 is_poly_face_mesh_vertex (*ezdxf.entities.Vertex attribute*), 348
 is_polygon_mesh (*ezdxf.entities.Polyline attribute*), 345
 is_polygon_mesh_vertex (*ezdxf.entities.Vertex attribute*), 348
 is_started (*ezdxf.render.trace.LinearTrace attribute*), 538
 is_supported_dxf_attrib () (*ezdxf.entities.DXFEntity method*), 281
 is_text_direction_negative (*ezdxf.entities.MultiLeader.dxf attribute*), 336
 is_text_vertical_stacked () (*in module ezdxf.tools.text*), 497
 is_upside_down_text_angle () (*in module ezdxf.tools.text*), 497
 is_vertical (*ezdxf.math.ConstructionLine attribute*), 436
 is_vertical (*ezdxf.math.ConstructionRay attribute*), 435
 is_xref (*ezdxf.entities.Block attribute*), 242
 is_xref_overlay (*ezdxf.entities.Block attribute*), 242
 isclose () (*ezdxf.math.Vec3 method*), 430
 items (*ezdxf.entities.Linetype.dxf attribute*), 226
 items () (*ezdxf.entities.Dictionary method*), 374
 items () (*ezdxf.entities.xdict.ExtensionDict method*), 390

items () (*ezdxf.entitydb.EntityDB method*), 687
 items () (*ezdxf.gfxattribs.GfxAttribs method*), 491
 iter_col () (*ezdxf.math.Matrix method*), 458
 iter_diag () (*ezdxf.math.Matrix method*), 458
 iter_row () (*ezdxf.math.Matrix method*), 458
 IterDXF (*class in ezdxf.addons.iterdxf*), 557
 IterDXFWriter (*class in ezdxf.addons.iterdxf*), 558

J
 join_style (*ezdxf.addons.acadctb.PlotStyle attribute*), 577
 julian_day (*ezdxf.entities.Sun.dxf attribute*), 385
 juliandate () (*in module ezdxf.tools*), 489
 justification (*ezdxf.entities.MLine.dxf attribute*), 322
 JUSTIFIED (*ezdxf.enums.MTextParagraphAlignment attribute*), 398
 JUSTIFIED (*ezdxf.tools.text.ezdxflldxf.const.MTextParagraphAlignment attribute*), 495

K
 key (*ezdxf.entities.DXFClass attribute*), 211
 key () (*ezdxf.sections.table.Table static method*), 216
 keys () (*ezdxf.entities.Dictionary method*), 374
 keys () (*ezdxf.entities.xdict.ExtensionDict method*), 390
 keys () (*ezdxf.entitydb.EntityDB method*), 687
 Kilometers (*ezdxf.enums.InsertUnits attribute*), 400
 knot_count () (*ezdxf.entities.Spline method*), 356
 knot_refinement () (*ezdxf.math.BSpline method*), 449
 knot_tolerance (*ezdxf.entities.Spline.dxf attribute*), 355
 knot_values (*ezdxf.entities.SplineEdge attribute*), 312
 knots (*ezdxf.entities.Spline attribute*), 356
 knots () (*ezdxf.math.BSpline method*), 448

L
 landing_gap (*ezdxf.entities.MLeaderStyle.dxf attribute*), 382
 landing_gap_size (*ezdxf.entities.MLeaderContext attribute*), 339
 last (*ezdxf.query.EntityQuery attribute*), 405
 last_height (*ezdxf.entities.Textstyle.dxf attribute*), 225
 last_leader_point (*ezdxf.entities.LeaderData attribute*), 340
 Layer (*class in ezdxf.entities*), 221
 layer (*ezdxf.entities.Block.dxf attribute*), 241
 layer (*ezdxf.entities.DXFGraphic.dxf attribute*), 286
 layer (*ezdxf.entities.EndBlk.dxf attribute*), 242
 layers (*ezdxf.addons.dxf2code.Code attribute*), 554
 layers (*ezdxf.document.Drawing attribute*), 200

layers (`ezdxf.sections.tables.TablesSection` attribute), 211
LayerTable (*class in* `ezdxf.sections.table`), 217
Layout (*class in* `ezdxf.layouts`), 274
layout (`ezdxf.entities.BlockRecord.dxf` attribute), 240
layout () (`ezdxf.document.Drawing` method), 202
layout_names () (`ezdxf.document.Drawing` method), 202
layout_names_in_taborder ()
 (`ezdxf.document.Drawing` method), 202
Layouts (*class in* `ezdxf.layouts`), 249
layouts (`ezdxf.document.Drawing` attribute), 200
layouts_and_blocks () (`ezdxf.document.Drawing` method), 204
Leader (*class in* `ezdxf.entities`), 315
leader_extend_to_text
 (`ezdxf.entities.MultiLeader.dxf` attribute), 336
leader_length (`ezdxf.entities.Dimension.dxf` attribute), 293
leader_line_color
 (`ezdxf.entities.MLeaderStyle.dxf` attribute), 382
leader_line_color (`ezdxf.entities.MultiLeader.dxf` attribute), 336
leader_linenotype_handle
 (`ezdxf.entities.MLeaderStyle.dxf` attribute), 382
leader_linenotype_handle
 (`ezdxf.entities.MultiLeader.dxf` attribute), 336
leader_lineweight
 (`ezdxf.entities.MLeaderStyle.dxf` attribute), 382
leader_lineweight (`ezdxf.entities.MultiLeader.dxf` attribute), 336
leader_offset_annotation_placement
 (`ezdxf.entities.Leader.dxf` attribute), 317
leader_offset_block_ref
 (`ezdxf.entities.Leader.dxf` attribute), 316
leader_point1 (`ezdxf.entities.ArcDimension.dxf` attribute), 300
leader_point2 (`ezdxf.entities.ArcDimension.dxf` attribute), 300
leader_type (`ezdxf.entities.MLeaderStyle.dxf` attribute), 382
leader_type (`ezdxf.entities.MultiLeader.dxf` attribute), 336
LeaderData (*class in* `ezdxf.entities`), 340
LeaderLine (*class in* `ezdxf.entities`), 340
leaders (`ezdxf.entities.MLeaderContext` attribute), 339
leading () (*in module* `ezdxf.tools.text`), 498
LEFT (`ezdxf.enums.MTextParagraphAlignment` attribute), 398
LEFT (`ezdxf.enums.TextEntityAlignment` attribute), 397
LEFT (`ezdxf.tools.text.ezdxflldxf.const.MTextParagraphAlignment` attribute), 495
left_attachment (`ezdxf.entities.MLeaderContext` attribute), 339
LEFT_TO_RIGHT (`ezdxf.enums.MTextFlowDirection` attribute), 399
legacy_repair () (`ezdxflldxf.extendedtags.ExtendedTags` method), 692
length (`ezdxf.entities.Linetype.dxf` attribute), 226
length () (`ezdxf.math.ConstructionLine` method), 436
LengthUnits (*class in* `ezdxf.enums`), 401
lens_length (`ezdxf.entities.View.dxf` attribute), 237
lens_length (`ezdxf.entities.VPort.dxf` attribute), 235
lerp () (`ezdxf.math.Vec3` method), 430
Lightyears (`ezdxf.enums.InsertUnits` attribute), 400
Line (*class in* `ezdxf.entities`), 317
LINE (`ezdxf.entities.EdgeType` attribute), 310
line_direction (`ezdxf.entities.MLineVertex` attribute), 324
line_params (`ezdxf.entities.MLineVertex` attribute), 324
line_policy (`ezdxf.addons.drawing.config.Configuration` attribute), 542
line_spacing_factor
 (`ezdxf.entities.Dimension.dxf` attribute), 294
line_spacing_factor (`ezdxf.entities.MText.dxf` attribute), 331
line_spacing_factor (`ezdxf.entities.MTextData` attribute), 341
line_spacing_style (`ezdxf.entities.Dimension.dxf` attribute), 294
line_spacing_style (`ezdxf.entities.MText.dxf` attribute), 331
line_spacing_style (`ezdxf.entities.MTextData` attribute), 341
line_to () (`ezdxf.path.Path` method), 473
linear_vertex_spacing ()
 (*in module* `ezdxf.math`), 415
LinearTrace (*class in* `ezdxf.render.trace`), 538
LineEdge (*class in* `ezdxf.entities`), 311
linepattern_size (`ezdxf.addons.acadctb.PlotStyle` attribute), 577
LinePolicy (*class in* `ezdxf.addons.drawing.config`), 543
lines (`ezdxf.entities.LeaderData` attribute), 340
lines (`ezdxf.entities.Pattern` attribute), 312
Linetype (*class in* `ezdxf.entities`), 226
linetype (`ezdxf.addons.acadctb.PlotStyle` attribute), 577
linetype (`ezdxf.entities.DXFGraphic.dxf` attribute), 286
linetype (`ezdxf.entities.ezdxfeentities.mline.MLineStyleElement` attribute), 325

linetype (*ezdxf.entities.Layer.dxf attribute*), 221
 linetypes (*ezdxf.addons.dx2code.Code attribute*), 554
 linetypes (*ezdxf.document.Drawing attribute*), 200
 linetypes (*ezdxf.sections.tables.TablesSection attribute*), 211
 LinetypeTable (*class in ezdxf.sections.table*), 217
 linewidth (*ezdxf.addons.acadctb.PlotStyle attribute*), 577
 linewidth (*ezdxf.entities.DXFGraphic.dxf attribute*), 286
 linewidth (*ezdxf.entities.Layer.dxf attribute*), 221
 linewidth_scaling (*ezdxf.addons.drawing.config.Configuration attribute*), 542
 lineweights (*ezdxf.addons.acadctb.ColorDependentPlotStyles attribute*), 575
 lineweights (*ezdxf.addons.acadctb.NamedPlotStyles attribute*), 576
 link_dxf_object () (*ezdxf.entities.Dictionary method*), 375
 link_dxf_object () (*ezdxf.entities.xdict.ExtensionDict method*), 391
 linspace () (*in module ezdxf.math*), 408
 list () (*ezdxf.math.Vec3 class method*), 429
 live_selection_handle (*ezdxf.entities.View.dxf attribute*), 238
 load () (*in module ezdxf.addons.acadctb*), 574
 load () (*in module ezdxf.tools.fonts*), 503
 load_from_header () (*ezdxf.gfxattribs.GfxAttribs class method*), 491
 load_proxy_graphics (*in module ezdxf.options*), 507
 loaded (*ezdxf.entities.ImageDef.dxf attribute*), 380
 loaded_config_files (*in module ezdxf.options*), 506
 local_cubic_bspline_interpolation () (*in module ezdxf.math*), 418
 location (*ezdxf.entities.MLineVertex attribute*), 324
 location (*ezdxf.entities.Point.dxf attribute*), 343
 location (*ezdxf.entities.Vertex.dxf attribute*), 348
 location (*ezdxf.math.ConstructionRay attribute*), 435
 lock () (*ezdxf.entities.Layer method*), 222
 LoftedSurface (*class in ezdxf.entities*), 358
 log_unprocessed_tags (*in module ezdxf.options*), 508
 lower (*ezdxf.math.BandedMatrixLU attribute*), 460
 lower_left (*ezdxf.entities.VPort.dxf attribute*), 235
 ltyscale (*ezdxf.entities.DXFGraphic.dxf attribute*), 286
 lu_decomp () (*ezdxf.math.Matrix method*), 459
 LUDecomposition (*class in ezdxf.math*), 459
 luminance () (*in module ezdxf.colors*), 402
 LWPolyline (*class in ezdxf.entities*), 320

M

m1 (*ezdxf.math.BandedMatrixLU attribute*), 460
 m2 (*ezdxf.math.BandedMatrixLU attribute*), 460
 m_close () (*ezdxf.entities.Polyline method*), 346
 m_count (*ezdxf.entities.Polyline.dxf attribute*), 345
 m_smooth_density (*ezdxf.entities.Polyline.dxf attribute*), 345
 MAGENTA (*ezdxf.enums.ACI attribute*), 402
 magnitude (*ezdxf.math.Vec3 attribute*), 428
 magnitude_square (*ezdxf.math.Vec3 attribute*), 428
 magnitude_xy (*ezdxf.math.Vec3 attribute*), 428
 main_axis_points () (*ezdxf.math.ConstructionEllipse method*), 443
 main_viewport () (*ezdxf.layouts.Paperspace method*), 277
 major_axis (*ezdxf.entities.Ellipse.dxf attribute*), 300
 major_axis (*ezdxf.math.ConstructionEllipse attribute*), 442
 major_axis_vector (*ezdxf.entities.EllipseEdge attribute*), 311
 make_font () (*ezdxf.entities.Textstyle method*), 226
 make_font () (*in module ezdxf.tools.fonts*), 500
 make_path () (*in module ezdxf.path*), 462
 make_primitive () (*in module ezdxf.disassemble*), 476
 map_to_globe () (*ezdxf.addons.geo.GeoProxy method*), 549
 material_handle (*ezdxf.entities.Layer.dxf attribute*), 222
 materials (*ezdxf.document.Drawing attribute*), 201
 MatplotlibFont (*class in ezdxf.tools.fonts*), 501
 Matrix (*class in ezdxf.math*), 457
 Matrix44 (*class in ezdxf.math*), 424
 max_flattening_distance
 (*ezdxf.addons.drawing.config.Configuration attribute*), 542
 max_flattening_distance
 (*ezdxf.disassemble.Primitive attribute*), 476
 max_leader_segments_points
 (*ezdxf.entities.MLeaderStyle.dxf attribute*), 383
 mcount (*ezdxf.entities.Insert attribute*), 243
 Measurement (*class in ezdxf.enums*), 400
 measurement (*ezdxf.addons.drawing.config.Configuration attribute*), 542
 measurement (*ezdxf.tools.fonts.AbstractFont attribute*), 500
 MengerSponge (*class in ezdxf.addons*), 583
 merge () (*ezdxf.addons.dx2code.Code method*), 555
 Mesh (*class in ezdxf.entities*), 326
 mesh () (*ezdxf.addons.MengerSponge method*), 583
 mesh () (*ezdxf.addons.pycsg.CSG method*), 573
 mesh () (*ezdxf.addons.SierpinskyPyramid method*), 588

mesh_faces_count (*ezdxf.entities.GeoData.dxf attribute*), 378
MeshAverageVertexMerger (class in *ezdxf.render*), 537
MeshBuilder (class in *ezdxf.render*), 533
MeshData (class in *ezdxf.entities*), 326
MeshTransformer (class in *ezdxf.render*), 535
MeshVertexCache (class in *ezdxf.entities*), 349
MeshVertexMerger (class in *ezdxf.render*), 536
Meters (*ezdxf.enums.InsertUnits attribute*), 400
Metric (*ezdxf.enums.Measurement attribute*), 400
Microinches (*ezdxf.enums.InsertUnits attribute*), 400
Microns (*ezdxf.enums.InsertUnits attribute*), 400
MIDDLE (*ezdxf.enums.MTextLineAlignment attribute*), 399
MIDDLE (*ezdxf.enums.TextEntityAlignment attribute*), 398
MIDDLE_CENTER (*ezdxf.enums.MTextEntityAlignment attribute*), 398
MIDDLE_CENTER (*ezdxf.enums.TextEntityAlignment attribute*), 398
MIDDLE_LEFT (*ezdxf.enums.MTextEntityAlignment attribute*), 398
MIDDLE_LEFT (*ezdxf.enums.TextEntityAlignment attribute*), 398
MIDDLE_RIGHT (*ezdxf.enums.MTextEntityAlignment attribute*), 398
MIDDLE_RIGHT (*ezdxf.enums.TextEntityAlignment attribute*), 398
midpoint () (*ezdxf.math.ConstructionLine method*), 436
Miles (*ezdxf.enums.InsertUnits attribute*), 400
Millimeters (*ezdxf.enums.InsertUnits attribute*), 400
Mils (*ezdxf.enums.InsertUnits attribute*), 400
min_dash_length (*ezdxf.addons.drawing.config.Configuration attribute*), 542
min_lineweight (*ezdxf.addons.drawing.config.Configuration attribute*), 542
minor_axis (*ezdxf.entities.Ellipse attribute*), 301
minor_axis (*ezdxf.math.ConstructionEllipse attribute*), 442
minor_axis_length (*ezdxf.entities.EllipseEdge attribute*), 311
misses (*ezdxf.bbox.Cache attribute*), 479
miter_direction (ezdxf.entities.MLineVertex attribute), 324
mleader_styles (ezdxf.document.Drawing attribute), 201
MLeaderContext (class in *ezdxf.entities*), 339
MLeaderStyle (class in *ezdxf.entities*), 381
MLine (class in *ezdxf.entities*), 322
mline_styles (ezdxf.document.Drawing attribute), 201
MLineStyle (class in *ezdxf.entities*), 324
MLineVertex (class in *ezdxf.entities*), 324
model_type () (*ezdxf.layouts.Layout method*), 276
Modelspace (class in *ezdxf.layouts*), 276
modelspace () (ezdxf.addons.iterdxf.IterDXF method), 558
modelspace () (ezdxf.document.Drawing method), 202
modelspace () (*ezdxf.layouts.Layouts method*), 249
modelspace () (in module *ezdxf.addons.iterdxf*), 557
monochrome (*ezdxf.entities.Underlay attribute*), 367
MonospaceFont (class in *ezdxf.tools.fonts*), 500
move_to () (*ezdxf.path.Path method*), 473
move_to_layout () (ezdxf.entities.DXFGraphic method), 285
move_to_layout () (ezdxf.layouts.BaseLayout method), 252
moveto () (*ezdxf.math.UCS method*), 423
MPolygon (class in *ezdxf.entities*), 327
MSLIDE (*ezdxf.enums.SortEntities attribute*), 401
MText (class in *ezdxf.entities*), 330
mtext (*ezdxf.entities.MLeaderContext attribute*), 339
mtext_size () (in module *ezdxf.tools.text_size*), 499
MTextBackgroundColor (class in *ezdxf.enums*), 399
MTextData (class in *ezdxf.entities*), 341
MTextEditor (class in *ezdxf.tools.text*), 492
MTextEntityAlignment (class in *ezdxf.enums*), 398
MTextExplode (class in *ezdxf.addons*), 567
MTextFlowDirection (class in *ezdxf.enums*), 399
MTextLineAlignment (class in *ezdxf.enums*), 399
MTextLineSpacing (class in *ezdxf.enums*), 399
MTextParagraphAlignment (class in *ezdxf.enums*), 398
MTextStroke (class in *ezdxf.enums*), 399
Multi-Path, 462
multi_insert () (ezdxf.entities.Insert method), 246
multi_path_from_matplotlib_path () (in module *ezdxf.path*), 463
multi_path_from_qpainter_path () (in module *ezdxf.path*), 463
multi_recursive () (in module *ezdxf.bbox*), 477
MultiLeader (class in *ezdxf.entities*), 335
MultiLeaderBlockBuilder (class in *ezdxf.render*), 540
MultiLeaderBuilder (class in *ezdxf.render*), 540
MultiLeaderMTextBuilder (class in *ezdxf.render*), 540

N

n_close () (*ezdxf.entities.Polyline method*), 346
n_control_points (*ezdxf.entities.Spline.dxf attribute*), 355
n_count (*ezdxf.entities.Polyline.dxf attribute*), 345

n_fit_points (*ezdxf.entities.Spline.dxf attribute*), 355
n_knots (*ezdxf.entities.Spline.dxf attribute*), 355
n_seed_points (*ezdxf.entities.Hatch.dxf attribute*), 303
n_smooth_density (*ezdxf.entities.Polyline.dxf attribute*), 345
name (*ezdxf.entities.AppID.dxf attribute*), 238
name (*ezdxf.entities.Block.dxf attribute*), 241
name (*ezdxf.entities.BlockRecord.dxf attribute*), 240
name (*ezdxf.entities.DimStyle.dxf attribute*), 227
name (*ezdxf.entities.DXFClass.dxf attribute*), 210
name (*ezdxf.entities.DXFLayout.dxf attribute*), 376
name (*ezdxf.entities.Insert.dxf attribute*), 243
name (*ezdxf.entities.Layer.dxf attribute*), 221
name (*ezdxf.entities.Linetype.dxf attribute*), 226
name (*ezdxf.entities.MLeaderStyle.dxf attribute*), 383
name (*ezdxf.entities.MLineStyle.dxf attribute*), 324
name (*ezdxf.entities.Shape.dxf attribute*), 352
name (*ezdxf.entities.Textstyle.dxf attribute*), 225
name (*ezdxf.entities.UCSTableEntry.dxf attribute*), 239
name (*ezdxf.entities.UnderlayDefinition.dxf attribute*), 386
name (*ezdxf.entities.View.dxf attribute*), 236
name (*ezdxf.entities.VPort.dxf attribute*), 235
name (*ezdxf.layouts.Layout attribute*), 274
name (*ezdxf.layouts.Modelspace attribute*), 276
name (*ezdxf.layouts.Paperspace attribute*), 277
name (*ezdxf.urecord.UserRecord attribute*), 487
NamedPlotStyles (*class in ezdxf.addons.acadctb*), 576
names () (*ezdxf.layouts.Layouts method*), 249
names_in_taborder () (*ezdxf.layouts.Layouts method*), 249
Nanometers (*ezdxf.enums.InsertUnits attribute*), 400
ncols (*ezdxf.math.BezierSurface attribute*), 454
ncols (*ezdxf.math.Matrix attribute*), 457
new () (*ezdxf.entities.dxfgroups.GroupCollection method*), 280
new () (*ezdxf.layouts.Layouts method*), 249
new () (*ezdxf.sections.blocks.BlocksSection method*), 212
new () (*ezdxf.sections.table.Table method*), 216
new () (*in module ezdxf*), 194
new () (*in module ezdxf.query*), 406
new_anonymous_block ()
 (*ezdxf.sections.blocks.BlocksSection method*), 212
new_app_data () (*ezdxf.lldx.dxf_extendedtags.ExtendedTags method*), 693
new_ctb () (*in module ezdxf.addons.acadctb*), 574
new_extension_dict () (*ezdxf.entities.DXFEntity method*), 282
new_geodata () (*ezdxf.layouts.Modelspace method*), 276
new_layout () (*ezdxf.document.Drawing method*), 202
new_stb () (*in module ezdxf.addons.acadctb*), 574
new_style () (*ezdxf.addons.acadctb.ColorDependentPlotStyles method*), 575
new_style () (*ezdxf.addons.acadctb.NamedPlotStyles method*), 576
new_trashcan () (*ezdxf.entitydb.EntityDB method*), 687
new_xdata () (*ezdxf.lldx.dxf_extendedtags.ExtendedTags method*), 693
next_handle () (*ezdxf.entitydb.EntityDB method*), 687
ngon () (*in module ezdxf.path*), 472
ngon () (*in module ezdxf.render.forms*), 529
no_twist (*ezdxf.entities.LoftedSurface.dxf attribute*), 359
noclass (*ezdxf.lldx.dxf_extendedtags.ExtendedTags attribute*), 692
NONE_TAG (*in module ezdxf.lldx.types*), 690
normal (*ezdxf.math.Plane attribute*), 432
normal_vector (*ezdxf.entities.Leader.dxf attribute*), 316
normal_vector_3p () (*in module ezdxf.math*), 415
normalize () (*ezdxf.math.Vec3 method*), 430
normalize_text_angle () (*in module ezdxf.tools*), 489
north_direction (*ezdxf.entities.GeoData.dxf attribute*), 377
nrows (*ezdxf.math.BandedMatrixLU attribute*), 460
nrows (*ezdxf.math.BezierSurface attribute*), 454
nrows (*ezdxf.math.LUDecomposition attribute*), 459
nrows (*ezdxf.math.Matrix attribute*), 457
NULLVEC (*in module ezdxf.math*), 431

O

OBJECT_COLOR (*in module ezdxf.addons.acadctb*), 578
OBJECT_COLOR2 (*in module ezdxf.addons.acadctb*), 578
OBJECT_LINETYPE (*in module ezdxf.addons.acadctb*), 578
OBJECT_LINEWEIGHT (*in module ezdxf.addons.acadctb*), 578
objects (*ezdxf.document.Drawing attribute*), 200
objects () (*in module ezdxf.zoom*), 510
ObjectsSection (*class in ezdxf.sections.objects*), 213
oblique (*ezdxf.entities.Shape.dxf attribute*), 352
oblique (*ezdxf.entities.Text.dxf attribute*), 361
oblique (*ezdxf.entities.Textstyle.dxf attribute*), 225
oblique () (*ezdxf.tools.text.MTextEditor method*), 493
oblique_angle (*ezdxf.entities.Dimension.dxf attribute*), 294

observation_from_tag
 (*ezdxf.entities.GeoData.dxf attribute*), 378
observation_to_tag
 (*ezdxf.entities.GeoData.dxf attribute*), 378
OCS (*class in ezdxf.math*), 421
ocs () (*ezdxf.entities.DXFGraphic method*), 284
OFF (*ezdxf.enums.MTextBackgroundColor attribute*), 399
off () (*ezdxf.entities.Layer method*), 223
offset (*ezdxf.entities.ezdxf.entities.mline.MLineStyleElement attribute*), 325
offset (*ezdxf.entities.PatternLine attribute*), 313
offset () (*ezdxf.math.Shape2d method*), 447
offset_vertices_2d () (*in module ezdxf.math*), 412
on (*ezdxf.entities.Underlay attribute*), 367
on () (*ezdxf.entities.Layer method*), 222
one_color (*ezdxf.entities.Gradient attribute*), 313
open_uniform_bspline () (*in module ezdxf.math*), 419
open_uniform_knot_vector () (*in module ezdxf.math*), 407
opendxf () (*in module ezdxf.addons.iterdxf*), 556
optimize () (*ezdxf.entities.MeshData method*), 327
optimize () (*ezdxf.entities.Polyface method*), 350
origin (*ezdxf.entities.UCSTableEntry.dxf attribute*), 239
orthogonal () (*ezdxf.math.ConstructionRay method*), 435
orthogonal () (*ezdxf.math.Vec3 method*), 429
outermost_paths () (*ezdxf.entities.BoundaryPaths method*), 306
output_encoding (*ezdxf.document.Drawing attribute*), 199
overlap () (*ezdxf.math.BoundingBox method*), 433
overlap () (*ezdxf.math.BoundingBox2d method*), 434
OVERLINE (*ezdxf.enums.MTextStroke attribute*), 399
overline () (*ezdxf.tools.text.MTextEditor method*), 493
override () (*ezdxf.entities.Dimension method*), 295
overwrite_property_value
 (*ezdxf.entities.MLeaderStyle.dxf attribute*), 383
owner (*ezdxf.entities.AppID.dxf attribute*), 238
owner (*ezdxf.entities.Block.dxf attribute*), 241
owner (*ezdxf.entities.BlockRecord.dxf attribute*), 240
owner (*ezdxf.entities.DimStyle.dxf attribute*), 227
owner (*ezdxf.entities.DXFEntity.dxf attribute*), 281
owner (*ezdxf.entities.EndBlk.dxf attribute*), 242
owner (*ezdxf.entities.Layer.dxf attribute*), 221
owner (*ezdxf.entities.Linetype.dxf attribute*), 226
owner (*ezdxf.entities.Textstyle.dxf attribute*), 225
owner (*ezdxf.entities.UCSTableEntry.dxf attribute*), 239
owner (*ezdxf.entities.View.dxf attribute*), 236
owner (*ezdxf.entities.VPort.dxf attribute*), 235

P

page_setup () (*ezdxf.layouts.Paperspace method*), 277
page_setup_name (*ezdxf.entities.PlotSettings.dxf attribute*), 385
Paperspace (*class in ezdxf.layouts*), 277
paperspace (*ezdxf.entities.DXFGraphic.dxf attribute*), 286
paragraph () (*ezdxf.tools.text.MTextEditor method*), 494
ParagraphProperties (*class in ezdxf.tools.text*), 495
param_t () (*ezdxf.math.ApproxParamT method*), 453
params () (*ezdxf.entities.Ellipse method*), 301
params () (*ezdxf.math.Bezier method*), 450
params () (*ezdxf.math.BSpline method*), 448
params () (*ezdxf.math.ConstructionEllipse method*), 443
params_from_vertices ()
 (*ezdxf.math.ConstructionEllipse method*), 443
parse () (*ezdxf.addons.geo.GeoProxy class method*), 547
Parsecs (*ezdxf.enums.InsertUnits attribute*), 400
Path (*class in ezdxf.path*), 473
path_entity_id (*ezdxf.entities.SweptSurface.dxf attribute*), 360
path_entity_transform_computed
 (*ezdxf.entities.ExtrudedSurface.dxf attribute*), 358
path_entity_transform_computed
 (*ezdxf.entities.SweptSurface.dxf attribute*), 360
path_entity_transformation_matrix
 (*ezdxf.entities.ExtrudedSurface attribute*), 358
path_entity_transformation_matrix ()
 (*ezdxf.entities.SweptSurface method*), 360
path_type (*ezdxf.entities.Leader.dxf attribute*), 316
path_type_flags
 (*ezdxf.entities.EdgePath attribute*), 309
path_type_flags
 (*ezdxf.entities.PolylinePath attribute*), 308
paths (*ezdxf.entities.BoundaryPaths attribute*), 306
paths (*ezdxf.entities.Hatch attribute*), 303
paths (*ezdxf.entities.MPolygon attribute*), 328
Pattern (*class in ezdxf.entities*), 312
pattern (*ezdxf.entities.Hatch attribute*), 303
pattern (*ezdxf.entities.MPolygon attribute*), 328
pattern_angle
 (*ezdxf.entities.Hatch.dxf attribute*), 303
pattern_angle
 (*ezdxf.entities.MPolygon.dxf attribute*), 328

pattern_double (*ezdxf.entities.Hatch.dxf attribute*), 303
 pattern_double (*ezdxf.entities.MPolygon.dxf attribute*), 328
 pattern_name (*ezdxf.entities.Hatch.dxf attribute*), 302
 pattern_name (*ezdxf.entities.MPolygon.dxf attribute*), 328
 pattern_scale (*ezdxf.entities.Hatch.dxf attribute*), 303
 pattern_scale (*ezdxf.entities.MPolygon.dxf attribute*), 328
 pattern_type (*ezdxf.entities.Hatch.dxf attribute*), 303
 pattern_type (*ezdxf.entities.MPolygon.dxf attribute*), 328
 PatternLine (*class in ezdxf.entities*), 312
 PdfDefinition (*class in ezdxf.entities*), 386
 PdfUnderlay (*class in ezdxf.entities*), 367
 pdmode (*ezdxf.addons.drawing.config.Configuration attribute*), 542
 pdsizs (*ezdxf.addons.drawing.config.Configuration attribute*), 541
 periodic (*ezdxf.entities.SplineEdge attribute*), 312
 perspective_lens_length
 (*ezdxf.entities.Viewport.dxf attribute*), 369
 perspective_projection()
 (*ezdxf.math.Matrix44 class method*), 426
 perspective_projection_fov()
 (*ezdxf.math.Matrix44 class method*), 426
 physical_pen_number
 (*ezdxf.addons.acadctb.PlotStyle attribute*), 577
 pixel_size (*ezdxf.entities.ImageDef.dxf attribute*), 380
 place() (*ezdxf.entities.Insert method*), 244
 Placeholder (*class in ezdxf.entities*), 384
 plain_mtext() (*ezdxf.entities.AttDef method*), 248
 plain_mtext() (*ezdxf.entities.Attrib method*), 247
 plain_mtext() (*in module ezdxf.tools.text*), 498
 plain_text() (*ezdxf.entities.MText method*), 333
 plain_text() (*ezdxf.entities.Text method*), 363
 plain_text() (*in module ezdxf.tools.text*), 498
 Plane (*class in ezdxf.math*), 432
 plane_normal_lofting_type
 (*ezdxf.entities.LoftedSurface.dxf attribute*), 359
 plane_normal_reversed
 (*ezdxf.entities.MLeaderContext attribute*), 339
 plane_origin (*ezdxf.entities.MLeaderContext attribute*), 339
 plane_x_axis (*ezdxf.entities.MLeaderContext attribute*), 339
 plane_y_axis (*ezdxf.entities.MLeaderContext attribute*), 339
 plot (*ezdxf.entities.Layer.dxf attribute*), 221
 PLOT (*ezdxf.enums.SortEntities attribute*), 401
 plot_centered() (*ezdxf.layouts.Layout method*), 276
 plot_flags_initializing()
 (*ezdxf.layouts.Layout method*), 276
 plot_hidden() (*ezdxf.layouts.Layout method*), 276
 plot_style_name (*ezdxf.entities.Viewport.dxf attribute*), 370
 plot_viewport_borders() (*ezdxf.layouts.Layout method*), 276
 PlotSettings (*class in ezdxf.entities*), 385
 PlotStyle (*class in ezdxf.addons.acadctb*), 577
 plotstyle_handle (*ezdxf.entities.Layer.dxf attribute*), 222
 Point (*class in ezdxf.entities*), 343
 point() (*ezdxf.math.Bezier method*), 451
 point() (*ezdxf.math.Bezier3P method*), 453
 point() (*ezdxf.math.Bezier4P method*), 452
 point() (*ezdxf.math.BezierSurface method*), 454
 point() (*ezdxf.math.BSpline method*), 449
 point() (*ezdxf.math.EulerSpiral method*), 454
 point_at() (*ezdxf.math.ConstructionCircle method*), 437
 point_to_line_relation() (*in module ezdxf.math*), 410
 points() (*ezdxf.entities.LWPolyline method*), 321
 points() (*ezdxf.entities.Polyline method*), 346
 points() (*ezdxf.math.Bezier method*), 451
 points() (*ezdxf.math.BSpline method*), 449
 points_from_wcs() (*ezdxf.math.OCS method*), 421
 points_from_wcs() (*ezdxf.math.UCS method*), 422
 points_to_ocs() (*ezdxf.math.UCS method*), 422
 points_to_wcs() (*ezdxf.math.OCS method*), 421
 points_to_wcs() (*ezdxf.math.UCS method*), 422
 Polyface (*class in ezdxf.entities*), 350
 Polyline (*class in ezdxf.entities*), 344
 POLYLINE (*ezdxf.entities.BoundaryPathType attribute*), 308
 polyline_to_edge_paths()
 (*ezdxf.entities.BoundaryPaths method*), 307
 PolylinePath (*class in ezdxf.entities*), 308
 Polymesh (*class in ezdxf.entities*), 349
 pop() (*ezdxf.entities.DimStyleOverride method*), 296
 pop_tags() (*ezdxf.lldx.tags.Tags method*), 692
 POSTSCRIPT (*ezdxf.enums.SortEntities attribute*), 401
 PREFER (*ezdxf.addons.drawing.config.ProxyGraphicPolicy attribute*), 544
 preserve_proxy_graphics() (*in module ezdxf.options*), 506
 prev_plot_init() (*ezdxf.layouts.Layout method*), 276
 Primitive (*class in ezdxf.disassemble*), 476
 print() (*in module ezdxf.options*), 505

print_lineweights() (*ezdxf.layouts.Layout method*), 276
project() (*ezdxf.math.Vec3 method*), 430
prompt (*ezdxf.entities.AttDef.dxf attribute*), 248
properties (*ezdxf.sections.header.CustomVars attribute*), 209
property_override_flags (*ezdxf.entities.MultiLeader.dxf attribute*), 337
proxy() (*in module ezdxf.addons.geo*), 546
proxy_graphic_policy (*ezdxf.addons.drawing.config.Configuration attribute*), 542
ProxyGraphicPolicy (*class in ezdxf.addons.drawing.config*), 543
purge() (*ezdxf.entitydb.EntityDB method*), 687
purge() (*ezdxf.entitydb.EntitySpace method*), 688
purge() (*ezdxf.layouts.BaseLayout method*), 251
pyramids() (*ezdxf.addons.SierpinskyPyramid method*), 588

Q

quadratic_bezier_from_3p() (*in module ezdxf.math*), 420
quadratic_to_cubic_bezier() (*in module ezdxf.math*), 420
query() (*ezdxf.document.Drawing method*), 202
query() (*ezdxf.layouts.BaseLayout method*), 251
query() (*ezdxf.query.EntityQuery method*), 405
query() (*ezdxf.sections.objects.ObjectsSection method*), 214

R

R12FastStreamWriter (*class in ezdxf.addons.r12writer*), 560
R12Spline (*class in ezdxf.render*), 525
r12writer() (*in module ezdxf.addons.r12writer*), 560
Radians (*ezdxf.enumsAngularUnits attribute*), 401
radius (*ezdxf.entities.Arc.dxf attribute*), 289
radius (*ezdxf.entities.ArcEdge attribute*), 311
radius (*ezdxf.entities.Circle.dxf attribute*), 291
radius (*ezdxf.entities.EllipseEdge attribute*), 311
radius (*ezdxf.math.ConstructionArc attribute*), 439
radius (*ezdxf.math.ConstructionCircle attribute*), 437
radius() (*ezdxf.math.EulerSpiral method*), 454
random_2d_path() (*in module ezdxf.render*), 527
random_3d_path() (*in module ezdxf.render*), 527
ratio (*ezdxf.entities.Ellipse.dxf attribute*), 300
ratio (*ezdxf.math.ConstructionEllipse attribute*), 442
rational (*ezdxf.entities.SplineEdge attribute*), 312
rational_bspline_from_arc() (*in module ezdxf.math*), 419
rational_bspline_from_ellipse() (*in module ezdxf.math*), 419
raw color, 700
Ray (*class in ezdxf.entities*), 351
ray (*ezdxf.math.ConstructionLine attribute*), 436
Reactors (*class in ezdxf.entities.appdata*), 698
read() (*in module ezdxf*), 196
read() (*in module ezdxf.recover*), 208
read_file() (*in module ezdxf.options*), 505
readfile() (*in module ezdxf*), 195
readfile() (*in module ezdxf.addons.odafc*), 565
readfile() (*in module ezdxf.recover*), 207
readzip() (*in module ezdxf*), 196
rebuild() (*ezdxf.blkrefs.BlockDefinitionIndex method*), 392
recreate_source_layout() (*ezdxf.addons.importer.Importer method*), 553
rect() (*in module ezdxf.path*), 472
rect_vertices() (*ezdxf.math.BoundingBox method*), 433
rect_vertices() (*ezdxf.math.BoundingBox2d method*), 434
recursive_decompose() (*in module ezdxf.disassemble*), 475
RED (*ezdxf.enums.AC1 attribute*), 402
REDRAW (*ezdxf.enums.SortEntities attribute*), 401
ref_vp_object_1 (*ezdxf.entities.Viewport.dxf attribute*), 372
ref_vp_object_2 (*ezdxf.entities.Viewport.dxf attribute*), 372
ref_vp_object_3 (*ezdxf.entities.Viewport.dxf attribute*), 372
ref_vp_object_4 (*ezdxf.entities.Viewport.dxf attribute*), 372
reference_point (*ezdxf.entities.GeoData.dxf attribute*), 377
reference_vector_for_controlling_twist (*ezdxf.entities.ExtrudedSurface.dxf attribute*), 358
reference_vector_for_controlling_twist (*ezdxf.entities.SweptSurface.dxf attribute*), 360
REGEN (*ezdxf.enums.SortEntities attribute*), 401
Region (*class in ezdxf.entities*), 351
register() (*ezdxf.sections.classes.ClassesSection method*), 210
reliable CAD application, 700
remove() (*ezdxf.entities.Dictionary method*), 375
remove() (*ezdxf.entitydb.EntitySpace method*), 688
remove() (*ezdxf.query.EntityQuery method*), 405
remove() (*ezdxf.sections.header.CustomVars method*), 210
remove() (*ezdxf.sections.table.Table method*), 216
remove_association() (*ezdxf.entities.Hatch method*), 305
remove_tags() (*ezdxf.lldxftags.Tags method*), 691

remove_tags_except() (*ezdxf.lldx.tags.Tags method*), 691
 rename() (*ezdxf.entities.Layer method*), 223
 rename() (*ezdxf.layouts.Layouts method*), 249
 rename_block() (*ezdxf.sections.blocks.BlocksSection method*), 212
 render() (*ezdxf.addons.MengerSponge method*), 583
 render() (*ezdxf.addons.SierpinskyPyramid method*), 588
 render() (*ezdxf.entities.Dimension method*), 295
 render() (*ezdxf.entities.DimStyleOverride method*), 299
 render() (*ezdxf.render.Bezier method*), 526
 render() (*ezdxf.render.R12Spline method*), 525
 render_3dfaces() (*ezdxf.render.MeshBuilder method*), 534
 render_as_fit_points() (*ezdxf.render.Spline method*), 523
 render_axis() (*ezdxf.math.OCS method*), 421
 render_axis() (*ezdxf.math.UCS method*), 424
 render_closed_bspline() (*ezdxf.render.Spline method*), 524
 render_closed_rbspline() (*ezdxf.render.Spline method*), 524
 render_hatches() (*in module ezdxf.path*), 464
 render_lines() (*in module ezdxf.path*), 465
 render_lwpolylines() (*in module ezdxf.path*), 463
 render_mesh() (*ezdxf.render.MeshBuilder method*), 534
 render_mode (*ezdxf.entities.View.dxf attribute*), 237
 render_mode (*ezdxf.entities.Viewport.dxf attribute*), 370
 render_mpolygons() (*in module ezdxf.path*), 464
 render_normals() (*ezdxf.render.MeshBuilder method*), 534
 render_open_bspline() (*ezdxf.render.Spline method*), 524
 render_open_rbspline() (*ezdxf.render.Spline method*), 524
 render_polyface() (*ezdxf.render.MeshBuilder method*), 534
 render_polyline() (*ezdxf.render.EulerSpiral method*), 526
 render_polylines2d() (*in module ezdxf.path*), 463
 render_polylines3d() (*in module ezdxf.path*), 465
 render_spline() (*ezdxf.render.EulerSpiral method*), 527
 renderSplines_and_polylines() (*in module ezdxf.path*), 465
 render_uniform_bspline() (*ezdxf.render.Spline method*), 524
 render_uniform_rbspline() (*ezdxf.render.Spline method*), 524
 rendering_paths() (*ezdxf.entities.BoundaryPaths method*), 306
 replace() (*ezdxf.math.Vec3 method*), 429
 replace() (*ezdxf.sections.header.CustomVars method*), 210
 replace_handle() (*ezdxf.lldx.extendedtags.ExtendedTags method*), 692
 replace_handle() (*ezdxf.lldx.tags.Tags method*), 690
 replace_xdata_list() (*ezdxf.entities.DXFEntity method*), 283
 replace_xlist() (*ezdxf.entities.xdata.XData method*), 696
 required_knot_values() (*in module ezdxf.math*), 407
 reset() (*ezdxf.entities.XRecord method*), 388
 reset() (*in module ezdxf.options*), 505
 reset_boundary_path() (*ezdxf.entities.Image method*), 315
 reset_boundary_path() (*ezdxf.entities.Underlay method*), 367
 reset_extents() (*ezdxf.layouts.Layout method*), 275
 reset_fingerprint_guid() (*ezdxf.document.Drawing method*), 204
 reset_limits() (*ezdxf.layouts.Layout method*), 275
 reset_main_viewport() (*ezdxf.layouts.Paperspace method*), 278
 reset_paper_limits() (*ezdxf.layouts.Paperspace method*), 278
 reset_transformation() (*ezdxf.entities.Insert method*), 245
 reset_version_guid() (*ezdxf.document.Drawing method*), 204
 reset_viewports() (*ezdxf.layouts.Paperspace method*), 278
 reshape() (*ezdxf.math.Matrix static method*), 458
 resolution_units (*ezdxf.entities.ImageDef.dxf attribute*), 380
 reverse() (*ezdxf.math.Bezier method*), 450
 reverse() (*ezdxf.math.Bezier3P method*), 452
 reverse() (*ezdxf.math.Bezier4P method*), 451
 reverse() (*ezdxf.math.BSpline method*), 448
 reversed() (*ezdxf.math.Vec3 method*), 430
 reversed() (*ezdxf.path.Path method*), 473
 revolve_angle (*ezdxf.entities.RevolvedSurface.dxf attribute*), 359
 RevolvedSurface (*class in ezdxf.entities*), 359
 rgb (*ezdxf.entities.DXFGraphic attribute*), 284
 rgb (*ezdxf.entities.Layer attribute*), 222
 rgb() (*ezdxf.tools.text.MTextEditor method*), 493
 rgb2int() (*in module ezdxf.colors*), 402
 RIGHT (*ezdxf.enums.MTextParagraphAlignment attribute*), 398
 RIGHT (*ezdxf.enums.TextEntityAlignment attribute*), 398

RIGHT (*ezdxf.tools.text.ezdxflldxf.const.MTextParagraphAlign*)
 attribute), 495
right_attachment (*ezdxf.entities.MLeaderContext*
 attribute), 339
rootdict (*ezdxf.document.Drawing* attribute), 200
rootdict (*ezdxf.sections.objects.ObjectsSection*
 attribute), 214
rotate() (*ezdxf.math.ConstructionBox* method), 445
rotate() (*ezdxf.math.Shape2d* method), 447
rotate() (*ezdxf.math.UCS* method), 423
rotate() (*ezdxf.math.Vec3* method), 431
rotate_axis() (*ezdxf.entities.DXFGraphic* method),
 285
rotate_axis() (*ezdxf.render.MeshTransformer*
 method), 536
rotate_deg() (*ezdxf.math.Vec3* method), 431
rotate_local_x() (*ezdxf.math.UCS* method), 423
rotate_local_y() (*ezdxf.math.UCS* method), 423
rotate_local_z() (*ezdxf.math.UCS* method), 423
rotate_rad() (*ezdxf.math.Shape2d* method), 447
rotate_x() (*ezdxf.entities.DXFGraphic* method), 285
rotate_x() (*ezdxf.render.MeshTransformer* method),
 536
rotate_y() (*ezdxf.entities.DXFGraphic* method), 285
rotate_y() (*ezdxf.render.MeshTransformer* method),
 536
rotate_z() (*ezdxf.entities.DXFGraphic* method), 285
rotate_z() (*ezdxf.math.ConstructionArc* method),
 440
rotate_z() (*ezdxf.render.MeshTransformer* method),
 536
rotation (*ezdxf.entities.BlockData* attribute), 342
rotation (*ezdxf.entities.Gradient* attribute), 313
rotation (*ezdxf.entities.Insert.dxf* attribute), 243
rotation (*ezdxf.entities.MText.dxf* attribute), 331
rotation (*ezdxf.entities.MTextData* attribute), 341
rotation (*ezdxf.entities.Shape.dxf* attribute), 352
rotation (*ezdxf.entities.Text.dxf* attribute), 361
rotation (*ezdxf.entities.Underlay.dxf* attribute), 366
rotation_form() (in module *ezdxf.render.forms*),
 532
row() (*ezdxf.math.Matrix* method), 458
row_count (*ezdxf.entities.Insert.dxf* attribute), 243
row_spacing (*ezdxf.entities.Insert.dxf* attribute), 243
rows() (*ezdxf.math.Matrix* method), 458
rows() (*ezdxf.math.Matrix44* method), 427
ruled_surface (*ezdxf.entities.LoftedSurface.dxf* at-
 tribute), 359
rytz_axis_construction() (in module
 ezdxf.math), 412

S

SAB, 700
safe_string() (in module *ezdxf.tools.text*), 498

save() (*ezdxf.addons.acadctb.ColorDependentPlotStyles*
 method), 575
save() (*ezdxf.addons.acadctb.NamedPlotStyles*
 method), 577
save() (*ezdxf.document.Drawing* method), 201
save() (in module *ezdxf.tools.fonts*), 503
saveas() (*ezdxf.document.Drawing* method), 201
scale (*ezdxf.entities.BlockData* attribute), 342
scale (*ezdxf.entities.BlockRecord.dxf* attribute), 240
scale (*ezdxf.entities.MLeaderContext* attribute), 339
scale (*ezdxf.entities.MLeaderStyle.dxf* attribute), 383
scale (*ezdxf.entities.MultiLeader.dxf* attribute), 337
scale (*ezdxf.entities.Underlay* attribute), 367
scale() (*ezdxf.entities.DXFGraphic* method), 285
scale() (*ezdxf.entities.Pattern* method), 312
scale() (*ezdxf.math.ConstructionBox* method), 445
scale() (*ezdxf.math.Matrix44* class method), 425
scale() (*ezdxf.math.Shape2d* method), 447
scale() (*ezdxf.render.MeshTransformer* method), 535
scale() (*ezdxf.tools.fonts.FontMeasurements* method),
 502
scale_estimation_method
 (*ezdxf.entities.GeoData.dxf* attribute), 378
scale_factor (*ezdxf.addons.acadctb.ColorDependentPlotStyles*
 attribute), 575
scale_factor (*ezdxf.addons.acadctb.NamedPlotStyles*
 attribute), 576
scale_factor (*ezdxf.entities.ExtrudedSurface.dxf* at-
 tribute), 358
scale_factor (*ezdxf.entities.MLine.dxf* attribute),
 322
scale_factor (*ezdxf.entities.SweptSurface.dxf* at-
 tribute), 360
scale_from_baseline()
 (*ezdxf.tools.fonts.FontMeasurements* method),
 502
scale_height() (*ezdxf.tools.text.MTextEditor*
 method), 493
scale_lineweights() (*ezdxf.layouts.Layout*
 method), 276
scale_uniform() (*ezdxf.entities.DXFGraphic*
 method), 285
scale_uniform() (*ezdxf.math.ConstructionArc*
 method), 440
scale_uniform() (*ezdxf.math.Shape2d* method), 447
scale_uniform() (*ezdxf.render.MeshTransformer*
 method), 535
scale_x (*ezdxf.entities.Underlay.dxf* attribute), 366
scale_y (*ezdxf.entities.Underlay.dxf* attribute), 366
scale_z (*ezdxf.entities.Underlay.dxf* attribute), 366
schema (*ezdxf.entities.DictionaryVar.dxf* attribute), 376
Scientific (*ezdxf.enums.LengthUnits* attribute), 401
screen (*ezdxf.addons.acadctb.PlotStyle* attribute), 577

sea_level_correction
 (*ezdxf.entities.GeoData.dxf attribute*), 378

sea_level_elevation (*ezdxf.entities.GeoData.dxf attribute*), 378

second_segment_angle_constraint
 (*ezdxf.entities.MLeaderStyle.dxf attribute*), 383

seeds (*ezdxf.entities.Hatch attribute*), 303

selectable (*ezdxf.entities.dxfgroups.DXFGGroup.dxf attribute*), 279

SELECTION (*ezdxf.enums.SortEntities attribute*), 401

set () (*ezdxf.entities.appdata.AppData method*), 697

set () (*ezdxf.entities.appdata.Reactors method*), 698

set () (*ezdxf.lldx.tags.VertexArray method*), 695

set () (*in module ezdxf.options*), 505

set_active_layout () (*ezdxf.layouts.Layouts method*), 250

set_align () (*ezdxf.entities.Text method*), 363

set_align_enum () (*ezdxf.entities.Text method*), 363

set_app_data () (*ezdxf.entities.DXFEntity method*), 282

set_app_data_content ()
 (*ezdxf.lldx.tags.ExtendedTags method*), 693

set_arrows () (*ezdxf.entities.DimStyle method*), 232

set_arrows () (*ezdxf.entities.DimStyleOverride method*), 296

set_axis () (*guide.ExampleCls method*), 699

set_bg_color () (*ezdxf.entities.MText method*), 332

set_boundary_path () (*ezdxf.entities.Image method*), 315

set_closed () (*ezdxf.entities.Spline method*), 356

set_closed_rational () (*ezdxf.entities.Spline method*), 357

set_col () (*ezdxf.math.Matrix method*), 458

set_col () (*ezdxf.math.Matrix44 method*), 425

set_color () (*ezdxf.entities.Layer method*), 223

set_data () (*ezdxf.entities.dxfgroups.DXFGGroup method*), 279

set_default () (*ezdxf.entities.DictionaryWithDefault method*), 376

set_diag () (*ezdxf.math.Matrix method*), 458

set_dimline_format () (*ezdxf.entities.DimStyle method*), 233

set_dimline_format ()
 (*ezdxf.entities.DimStyleOverride method*), 297

set_dxf_attrib () (*ezdxf.entities.DXFEntity method*), 282

set_extended_font_data ()
 (*ezdxf.entities.Textstyle method*), 226

set_extline1 () (*ezdxf.entities.DimStyle method*), 233

set_extline1 () (*ezdxf.entities.DimStyleOverride method*), 298

set_extline2 () (*ezdxf.entities.DimStyle method*), 233

set_extline2 () (*ezdxf.entities.DimStyleOverride method*), 298

set_extline_format () (*ezdxf.entities.DimStyle method*), 233

set_extline_format ()
 (*ezdxf.entities.DimStyleOverride method*), 298

set_first () (*ezdxf.lldx.tags.Tags method*), 691

set_flag_state () (*ezdxf.entities.DXFEntity method*), 282

set_flag_state () (*in module ezdxf.tools*), 489

set_gradient () (*ezdxf.entities.Hatch method*), 305

set_gradient () (*ezdxf.entities.MPolygon method*), 329

set_hyperlink () (*ezdxf.entities.DXFGraphic method*), 285

set_justification () (*ezdxf.entities.MLine method*), 323

set_limits () (*ezdxf.entities.DimStyle method*), 234

set_limits () (*ezdxf.entities.DimStyleOverride method*), 297

set_location () (*ezdxf.entities.DimStyleOverride method*), 298

set_location () (*ezdxf.entities.MText method*), 332

set_masking_area () (*ezdxf.entities.Wipeout method*), 372

set_mesh_vertex () (*ezdxf.entities.Polymesh method*), 349

set_modelspace_vport ()
 (*ezdxf.document.Drawing method*), 204

set_mtext () (*ezdxf.entities.AttDef method*), 248

set_mtext () (*ezdxf.entities.Attrib method*), 247

set_open_rational () (*ezdxf.entities.Spline method*), 356

set_open_uniform () (*ezdxf.entities.Spline method*), 356

set_pattern_angle () (*ezdxf.entities.Hatch method*), 304

set_pattern_angle () (*ezdxf.entities.MPolygon method*), 329

set_pattern_definition () (*ezdxf.entities.Hatch method*), 303

set_pattern_definition ()
 (*ezdxf.entities.MPolygon method*), 328

set_pattern_fill () (*ezdxf.entities.Hatch method*), 304

set_pattern_fill () (*ezdxf.entities.MPolygon method*), 329

set_pattern_scale () (*ezdxf.entities.Hatch method*), 304

set_pattern_scale () (*ezdxf.entities.MPolygon method*), 329

method), 329
set_placement() (ezdxf.entities.Text method), 362
set_plot_flags() (ezdxf.layouts.Layout method), 276
set_plot_style() (ezdxf.layouts.Layout method), 275
set_plot_type() (ezdxf.layouts.Layout method), 275
set_plot_window() (ezdxf.layouts.Layout method), 275
set_points() (ezdxf.entities.LWPolyline method), 321
set_pos() (ezdxf.entities.Text method), 362
set_raster_variables()
 (ezdxf.document.Drawing method), 203
set_raster_variables()
 (ezdxf.sections.objects.ObjectsSection method), 215
set_reactors() (ezdxf.entities.DXFEntity method), 284
set_redraw_order()
 (ezdxf.layouts.Layout method), 276
set_rotation() (ezdxf.entities.MText method), 332
set_row() (ezdxf.math.Matrix method), 458
set_row() (ezdxf.math.Matrix44 method), 425
set_scale() (ezdxf.entities.Insert method), 243
set_scale_factor()
 (ezdxf.entities.MLine method), 323
set_seed_points() (ezdxf.entities.Hatch method), 305
set_solid_fill() (ezdxf.entities.Hatch method), 304
set_solid_fill()
 (ezdxf.entities.MPolygon method), 329
set_style() (ezdxf.entities.MLine method), 323
set_table_lineweight()
 (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 575
set_table_lineweight()
 (ezdxf.addons.acadctb.NamedPlotStyles method), 576
set_text() (ezdxf.entities.Body method), 290
set_text() (ezdxf.entities.DimStyleOverride method), 298
set_text_align() (ezdxf.entities.DimStyle method), 232
set_text_align() (ezdxf.entities.DimStyleOverride method), 296
set_text_format()
 (ezdxf.entities.DimStyle method), 233
set_text_format()
 (ezdxf.entities.DimStyleOverride method), 297
set_tick() (ezdxf.entities.DimStyle method), 232
set_tick()
 (ezdxf.entities.DimStyleOverride method), 296
set_tolerance() (ezdxf.entities.DimStyle method), 234
set_tolerance()
 (ezdxf.entities.DimStyleOverride method), 296
set_transformation_matrix_lofted_entity
 (ezdxf.entities.LoftedSurface attribute), 359
set_underlay_def()
 (ezdxf.entities.Underlay method), 367
set_uniform() (ezdxf.entities.Spline method), 356
set_uniform_rational()
 (ezdxf.entities.Spline method), 356
set_values()
 (ezdxf.lldxp.packedtags.TagArray method), 694
set_vertices() (ezdxf.entities.Leader method), 317
set_vertices()
 (ezdxf.entities.PolylinePath method), 308
set_wipeout_variables()
 (ezdxf.document.Drawing method), 203
set_wipeout_variables()
 (ezdxf.sections.objects.ObjectsSection method), 216
set_xdata() (ezdxf.entities.DXFEntity method), 283
set_xdata()
 (ezdxf.lldxp.extendedtags.ExtendedTags method), 693
set_xdata_list()
 (ezdxf.entities.DXFEntity method), 283
set_xlist() (ezdxf.entities.XData method), 696
setup_local_grid()
 (ezdxf.entities.GeoData method), 379
shade_plot_handle (ezdxf.entities.Viewport.dxf attribute), 371
shade_plot_mode (ezdxf.entities.Viewport.dxf attribute), 371
shadow_map_size (ezdxf.entities.Sun.dxf attribute), 386
shadow_mode (ezdxf.entities.DXFGraphic.dxf attribute), 287
shadow_softness (ezdxf.entities.Sun.dxf attribute), 386
shadow_type (ezdxf.entities.Sun.dxf attribute), 385
shadows (ezdxf.entities.Sun.dxf attribute), 385
Shape (class in ezdxf.entities), 352
shape (ezdxf.math.Matrix attribute), 457
Shape2d (class in ezdxf.math), 447
shear_xy() (ezdxf.math.Matrix44 class method), 426
shift() (ezdxf.math.ucs method), 423
shift() (ezdxf.tools.fonts.FontMeasurements method), 502
shift_text()
 (ezdxf.entities.DimStyleOverride method), 298
SHOW (ezdxf.addons.drawing.config.ProxyGraphicPolicy

attribute), 543

`SHOW_APPROXIMATE_PATTERN (ezdxf.addons.drawing.config.HatchPolicy attribute), 543`

`show_defpoints (ezdxf.addons.drawing.config.Configuration attribute), 542`

`SHOW_OUTLINE (ezdxf.addons.drawing.config.HatchPolicy attribute), 543`

`show_plot_styles () (ezdxf.layouts.Layout method), 276`

`SHOW_SOLID (ezdxf.addons.drawing.config.HatchPolicy attribute), 543`

`SierpinskyPyramid (class in ezdxf.addons), 587`

`signed_distance_to () (ezdxf.math.Plane method), 432`

`simple_surfaces (ezdxf.entities.LoftedSurface.dxf attribute), 359`

`Single-Path, 462`

`single_pass_modelspace () (in module ezdxf.iterdxf), 557`

`single_paths () (in module ezdxf.path), 470`

`size (ezdxf.entities.Shape.dxf attribute), 352`

`slope (ezdxf.math.ConstructionRay attribute), 435`

`smooth_type (ezdxf.entities.Polyline.dxf attribute), 345`

`SNAP (ezdxf.enums.SortEntities attribute), 401`

`snap_angle (ezdxf.entities.Viewport.dxf attribute), 369`

`snap_base (ezdxf.entities.VPort.dxf attribute), 235`

`snap_base_point (ezdxf.entities.Viewport.dxf attribute), 369`

`snap_isopair (ezdxf.entities.VPort.dxf attribute), 236`

`snap_on (ezdxf.entities.VPort.dxf attribute), 236`

`snap_rotation (ezdxf.entities.VPort.dxf attribute), 236`

`snap_spacing (ezdxf.entities.Viewport.dxf attribute), 369`

`snap_spacing (ezdxf.entities.VPort.dxf attribute), 235`

`snap_style (ezdxf.entities.VPort.dxf attribute), 236`

`Solid (class in ezdxf.entities), 354`

`SOLID (ezdxf.addons.drawing.config.LinePolicy attribute), 543`

`solid (ezdxf.entities.ExtrudedSurface.dxf attribute), 358`

`solid (ezdxf.entities.LoftedSurface.dxf attribute), 359`

`solid (ezdxf.entities.RevolvedSurface.dxf attribute), 359`

`solid (ezdxf.entities.SweptSurface.dxf attribute), 360`

`Solid3d (class in ezdxf.entities), 288`

`solid_fill (ezdxf.entities.Hatch.dxf attribute), 302`

`solid_fill (ezdxf.entities.MPolygon.dxf attribute), 328`

`solve_matrix () (ezdxf.math.BandedMatrixLU method), 460`

`solve_matrix () (ezdxf.math.LUDecomposition`

`method), 460`

`solve_vector () (ezdxf.math.BandedMatrixLU method), 460`

`solve_vector () (ezdxf.math.LUDecomposition`

`SortEntities (class in ezdxf.enums), 401`

`source (ezdxf.addons.importer.Importer attribute), 551`

`source_boundary_objects (ezdxf.entities.EdgePath attribute), 309`

`source_boundary_objects (ezdxf.entities.PolylinePath attribute), 308`

`source_vertices (ezdxf.entities.GeoData attribute), 378`

`space_width () (ezdxf.tools.fonts.AbstractFont method), 500`

`space_width () (ezdxf.tools.fonts.MatplotlibFont method), 501`

`space_width () (ezdxf.tools.fonts.MonospaceFont method), 500`

`spatial_angle (ezdxf.math.Vec3 attribute), 429`

`spatial_angle_deg (ezdxf.math.Vec3 attribute), 429`

`sphere () (in module ezdxf.render.forms), 531`

`Spline (class in ezdxf.entities), 355`

`Spline (class in ezdxf.render), 523`

`SPLINE (ezdxf.entities.EdgeType attribute), 311`

`spline_edges_to_line_edges () (ezdxf.entities.BoundaryPaths method), 307`

`SplineEdge (class in ezdxf.entities), 311`

`split_bezier () (in module ezdxf.math), 421`

`square () (in module ezdxf.render.forms), 528`

`stack () (ezdxf.tools.text.MTextEditor method), 493`

`star () (in module ezdxf.path), 472`

`star () (in module ezdxf.render.forms), 529`

`start (ezdxf.entities.Line.dxf attribute), 317`

`start (ezdxf.entities.LineEdge attribute), 311`

`start (ezdxf.entities.Ray.dxf attribute), 351`

`start (ezdxf.entities.XLine.dxf attribute), 373`

`start (ezdxf.math.ConstructionEllipse attribute), 442`

`start (ezdxf.math.ConstructionLine attribute), 436`

`start () (ezdxf.render.Bezier method), 526`

`start_angle (ezdxf.entities.Arc.dxf attribute), 289`

`start_angle (ezdxf.entities.ArcDimension.dxf attribute), 300`

`start_angle (ezdxf.entities.ArcEdge attribute), 311`

`start_angle (ezdxf.entities.EllipseEdge attribute), 311`

`start_angle (ezdxf.entities.MLineStyle.dxf attribute), 325`

`start_angle (ezdxf.entities.RevolvedSurface.dxf attribute), 359`

`start_angle (ezdxf.math.ConstructionArc attribute), 439`

start_angle_rad (*ezdxf.math.ConstructionArc attribute*), 439
start_draft_angle (*ezdxf.entities.LoftedSurface.dxf attribute*), 359
start_draft_distance (*ezdxf.entities.RevolvedSurface.dxf attribute*), 359
start_draft_magnitude (*ezdxf.entities.LoftedSurface.dxf attribute*), 359
start_location (*ezdxf.entities.MLine.dxf attribute*), 323
start_location() (*ezdxf.entities.MLine method*), 323
start_param (*ezdxf.entities.Ellipse.dxf attribute*), 300
start_point (*ezdxf.entities.Arc attribute*), 289
start_point (*ezdxf.entities.Ellipse attribute*), 300
start_point (*ezdxf.math.ConstructionArc attribute*), 439
start_point (*ezdxf.math.ConstructionEllipse attribute*), 443
start_tangent (*ezdxf.entities.Spline.dxf attribute*), 355
start_tangent (*ezdxf.entities.SplineEdge attribute*), 312
start_width (*ezdxf.entities.Vertex.dxf attribute*), 348
status (*ezdxf.entities.Sun.dxf attribute*), 385
status (*ezdxf.entities.Viewport.dxf attribute*), 368
status (*ezdxf.entities.VPort.dxf attribute*), 236
STB, 700
store_proxy_graphics (*in module ezdxf.options*), 507
stretch (*ezdxf.tools.fonts.FontFace attribute*), 502
stretch() (*ezdxf.tools.text.TextLine method*), 496
STRIKE_THROUGH (*ezdxf.enums.MTextStroke attribute*), 399
strike_through() (*ezdxf.tools.text.MTextEditor method*), 493
strip() (*ezdxf.lldx.tags.Tags class method*), 692
style (*ezdxf.entities.MText.dxf attribute*), 331
style (*ezdxf.entities.Text.dxf attribute*), 361
style (*ezdxf.tools.fonts.FontFace attribute*), 501
style_element_count (*ezdxf.entities.MLine.dxf attribute*), 323
style_handle (*ezdxf.entities.MLine.dxf attribute*), 322
style_handle (*ezdxf.entities.MTextData attribute*), 341
style_handle (*ezdxf.entities.MultiLeader.dxf attribute*), 337
style_name (*ezdxf.entities.MLine.dxf attribute*), 322
styles (*ezdxf.addons.dxf2code.Code attribute*), 554
styles (*ezdxf.document.Drawing attribute*), 200
styles (*ezdxf.sections.tables.TablesSection attribute*), 211
sub_paths() (*ezdxf.path.Path method*), 473
subclasses (*ezdxf.lldx.tags.ExtendedTags attribute*), 692
subdivide() (*ezdxf.render.MeshTransformer method*), 535
subdivide() (*ezdxf.render.Spline method*), 523
subdivide_face() (*in module ezdxf.math*), 415
subdivide_ngons() (*in module ezdxf.math*), 416
subdivision_levels (*ezdxf.entities.Mesh.dxf attribute*), 326
subtract() (*ezdxf.addons.pycsg.CSG method*), 573
sum() (*ezdxf.math.Vec3 static method*), 431
Sun (*class in ezdxf.entities*), 385
sun_handle (*ezdxf.entities.View.dxf attribute*), 238
sun_handle (*ezdxf.entities.Viewport.dxf attribute*), 372
suppress_zeros() (*in module ezdxf.tools*), 489
Surface (*class in ezdxf.entities*), 357
swap_axis() (*ezdxf.math.ConstructionEllipse method*), 444
swap_cols() (*ezdxf.math.Matrix method*), 459
swap_rows() (*ezdxf.math.Matrix method*), 458
sweep_alignment (*ezdxf.entities.SweptSurface.dxf attribute*), 360
sweep_alignment_flags
 (*ezdxf.entities.ExtrudedSurface.dxf attribute*), 358
sweep_entity_transform_computed
 (*ezdxf.entities.ExtrudedSurface.dxf attribute*), 358
sweep_entity_transform_computed
 (*ezdxf.entities.SweptSurface.dxf attribute*), 360
sweep_entity_transformation_matrix
 (*ezdxf.entities.ExtrudedSurface attribute*), 358
sweep_entity_transformation_matrix()
 (*ezdxf.entities.SweptSurface method*), 360
sweep_vector (*ezdxf.entities.ExtrudedSurface.dxf attribute*), 358
swept_entity_id (*ezdxf.entities.SweptSurface.dxf attribute*), 360
SweptSurface (*class in ezdxf.entities*), 360

T

Table (*class in ezdxf.sections.table*), 216
table_entries_to_code() (*in module ezdxf.addons.dxf2code*), 554
tables (*ezdxf.document.Drawing attribute*), 200
TablesSection (*class in ezdxf.sections.tables*), 211
tag (*ezdxf.entities.AttDef.dxf attribute*), 248
tag (*ezdxf.entities.Attrib.dxf attribute*), 247
tag_index() (*ezdxf.lldx.tags.Tags method*), 691
TagArray (*class in ezdxf.lldx.packedtags*), 694

TagList (*class in ezdxf.lldxf.packedtags*), 694
 Tags (*class in ezdxf.lldxf.tags*), 690
 tags (*ezdxf.entities.XRecord attribute*), 387
 tangent (*ezdxf.entities.Vertex.dxf attribute*), 348
 tangent () (*ezdxf.math.Bezier3P method*), 453
 tangent () (*ezdxf.math.Bezier4P method*), 452
 tangent () (*ezdxf.math.ConstructionCircle method*), 437
 tangent () (*ezdxf.math.EulerSpiral method*), 454
 tangents () (*ezdxf.math.ConstructionArc method*), 439
 target (*ezdxf.addons.importer.Importer attribute*), 551
 target_point (*ezdxf.entities.View.dxf attribute*), 237
 target_point (*ezdxf.entities.VPort.dxf attribute*), 235
 target_vertices (*ezdxf.entities.GeoData attribute*), 378
 test_files (*in module ezdxf.options*), 507
 test_files_path (*in module ezdxf.options*), 507
 Text (*class in ezdxf.entities*), 361
 text (*ezdxf.entities.AttDef.dxf attribute*), 248
 text (*ezdxf.entities.Attrib.dxf attribute*), 247
 text (*ezdxf.entities.AttribData attribute*), 341
 text (*ezdxf.entities.Dimension.dxf attribute*), 294
 text (*ezdxf.entities.MText attribute*), 332
 text (*ezdxf.entities.Text.dxf attribute*), 361
 text (*ezdxf.tools.text.MTextEditor attribute*), 492
 text_align_always_left
 (*ezdxf.entities.MLeaderStyle.dxf attribute*), 383
 text_align_type
 (*ezdxf.entities.MLeaderContext attribute*), 339
 text_alignment_type
 (*ezdxf.entities.MLeaderStyle.dxf attribute*), 383
 text_alignment_type
 (*ezdxf.entities.MultiLeader.dxf attribute*), 337
 text_angle_type
 (*ezdxf.entities.MLeaderStyle.dxf attribute*), 383
 text_angle_type
 (*ezdxf.entities.MultiLeader.dxf attribute*), 337
 text_attachment_direction
 (*ezdxf.entities.MLeaderStyle.dxf attribute*), 383
 text_attachment_direction
 (*ezdxf.entities.MultiLeader.dxf attribute*), 337
 text_attachment_point
 (*ezdxf.entities.MultiLeader.dxf attribute*), 337
 text_bottom_attachment_type
 (*ezdxf.entities.MLeaderStyle.dxf attribute*), 383
 text_bottom_attachment_type
 (*ezdxf.entities.MultiLeader.dxf attribute*), 337
 (ezdxf.entities.MultiLeader.dxf attribute), 337
 text_color
 (*ezdxf.entities.MLeaderStyle.dxf attribute*), 383
 text_color
 (*ezdxf.entities.MultiLeader.dxf attribute*), 337
 text_direction
 (*ezdxf.entities.MText.dxf attribute*), 331
 text_direction
 (*ezdxf.entities.MTextData attribute*), 341
 text_generation_flag
 (*ezdxf.entities.Text.dxf attribute*), 362
 text_height
 (*ezdxf.entities.Leader.dxf attribute*), 316
 text_IPE_align
 (*ezdxf.entities.MultiLeader.dxf attribute*), 337
 text_left_attachment_type
 (*ezdxf.entities.MLeaderStyle.dxf attribute*), 384
 text_left_attachment_type
 (*ezdxf.entities.MultiLeader.dxf attribute*), 337
 text_midpoint
 (*ezdxf.entities.Dimension.dxf attribute*), 293
 text_right_attachment_type
 (*ezdxf.entities.MLeaderStyle.dxf attribute*), 384
 text_right_attachment_type
 (*ezdxf.entities.MultiLeader.dxf attribute*), 338
 text_rotation
 (*ezdxf.entities.Dimension.dxf attribute*), 294
 text_size () (*in module ezdxf.tools.text_size*), 499
 text_style_handle
 (*ezdxf.entities.MLeaderStyle.dxf attribute*), 384
 text_style_handle
 (*ezdxf.entities.MultiLeader.dxf attribute*), 338
 text_top_attachment_type
 (*ezdxf.entities.MLeaderStyle.dxf attribute*), 384
 text_top_attachment_type
 (*ezdxf.entities.MultiLeader.dxf attribute*), 338
 text_width
 (*ezdxf.entities.Leader.dxf attribute*), 316
 text_width ()
 (*ezdxf.tools.fonts.AbstractFont method*), 500
 text_width ()
 (*ezdxf.tools.fonts MatplotlibFont method*), 501
 text_width ()
 (*ezdxf.tools.fonts MonospaceFont method*), 500
 text_wrap () (*in module ezdxf.tools.text*), 498
 TextEntityAlignment (*class in ezdxf.enums*), 397
 TextLine (*class in ezdxf.tools.text*), 495
 Textstyle (*class in ezdxf.entities*), 225

TextstyleTable (*class in ezdxf.sections.table*), 218
thaw() (*ezdxf.entities.Layer method*), 222
thickness (*ezdxf.entities.DXFGraphic.dxf attribute*), 286
thickness (*ezdxf.entities.Line.dxf attribute*), 317
time (*ezdxf.entities.Sun.dxf attribute*), 385
tint (*ezdxf.entities.Gradient attribute*), 313
to_bsplines_and_vertices() (*in module ezdxf.path*), 468
to_control_vertices() (*in module ezdxf.disassemble*), 476
to_dxf_entities() (*ezdxf.addons.geo.GeoProxy method*), 547
to_ellipse() (*ezdxf.entities.Arc method*), 289
to_ellipse() (*ezdxf.entities.Circle method*), 291
to_hatches() (*in module ezdxf.path*), 466
to_lines() (*in module ezdxf.path*), 467
to_lwpolylines() (*in module ezdxf.path*), 466
to_matplotlib_path() (*in module ezdxf.path*), 468
to_meshes() (*in module ezdxf.disassemble*), 476
to_mpolygons() (*in module ezdxf.path*), 467
to_multi_path() (*in module ezdxf.path*), 470
to_ocs() (*ezdxf.math.ConstructionEllipse method*), 443
to_ocs() (*ezdxf.math.UCS method*), 422
to_ocs_angle_deg() (*ezdxf.math.UCS method*), 423
to_paths() (*in module ezdxf.disassemble*), 476
to_polylines2d() (*in module ezdxf.path*), 466
to_polylines3d() (*in module ezdxf.path*), 467
to_primitives() (*in module ezdxf.disassemble*), 476
to_qpainter_path() (*in module ezdxf.path*), 468
to_spline() (*ezdxf.entities.Arc method*), 289
to_spline() (*ezdxf.entities.Circle method*), 291
to_spline() (*ezdxf.entities.Ellipse method*), 301
to_splines_and_polylines() (*in module ezdxf.path*), 468
to_vertices() (*in module ezdxf.disassemble*), 476
to_wcs() (*ezdxf.math.OCS method*), 421
to_wcs() (*ezdxf.math.UCS method*), 422
tobytes() (*ezdxf.entities.Body method*), 290
TOP (*ezdxf.enums.MTextLineAlignment attribute*), 399
top_attachment (*ezdxf.entities.MLeaderContext attribute*), 340
TOP_CENTER (*ezdxf.enums.MTextEntityAlignment attribute*), 398
TOP_CENTER (*ezdxf.enums.TextEntityAlignment attribute*), 398
TOP_LEFT (*ezdxf.enums.MTextEntityAlignment attribute*), 398
TOP_LEFT (*ezdxf.enums.TextEntityAlignment attribute*), 398
TOP_RIGHT (*ezdxf.enums.MTextEntityAlignment attribute*), 398
TOP_RIGHT (*ezdxf.enums.TextEntityAlignment attribute*), 398
TOP_TO_BOTTOM (*ezdxf.enums.MTextFlowDirection attribute*), 399
tostring() (*ezdxf.entities.Body method*), 290
tostring() (*ezdxf.lldx.types.DXFBinaryTag method*), 689
tostring() (*ezdxf.tools.text.ParagraphProperties method*), 495
total_height (*ezdxf.tools.text_size.ezdxf.tools.text_size.MTextSize attribute*), 499
total_height (*ezdxf.tools.text_size.ezdxf.tools.text_size.TextSize attribute*), 499
total_width (*ezdxf.tools.text_size.ezdxf.tools.text_size.MTextSize attribute*), 499
Trace (*class in ezdxf.entities*), 365
TraceBuilder (*class in ezdxf.render.trace*), 537
transform() (*ezdxf.entities.Arc method*), 289
transform() (*ezdxf.entities.Circle method*), 291
transform() (*ezdxf.entities.Dimension method*), 295
transform() (*ezdxf.entities.DXFGraphic method*), 285
transform() (*ezdxf.entities.Ellipse method*), 301
transform() (*ezdxf.entities.Face3d method*), 288
transform() (*ezdxf.entities.Hatch method*), 305
transform() (*ezdxf.entities.Image method*), 315
transform() (*ezdxf.entities.Insert method*), 245
transform() (*ezdxf.entities.Leader method*), 317
transform() (*ezdxf.entities.Line method*), 318
transform() (*ezdxf.entities.LWPolyline method*), 322
transform() (*ezdxf.entities.Mesh method*), 326
transform() (*ezdxf.entities.MLine method*), 324
transform() (*ezdxf.entities.MPolygon method*), 330
transform() (*ezdxf.entities.MText method*), 333
transform() (*ezdxf.entities.MultiLeader method*), 339
transform() (*ezdxf.entities.Point method*), 343
transform() (*ezdxf.entities.Polyline method*), 347
transform() (*ezdxf.entities.Ray method*), 351
transform() (*ezdxf.entities.Shape method*), 352
transform() (*ezdxf.entities.Solid method*), 354
transform() (*ezdxf.entities.Spline method*), 357
transform() (*ezdxf.entities.Text method*), 363
transform() (*ezdxf.entities.Trace method*), 365
transform() (*ezdxf.entities.xdata.XData method*), 697
transform() (*ezdxf.entities.XLine method*), 373
transform() (*ezdxf.math.Bezier method*), 450
transform() (*ezdxf.math.Bezier3P method*), 452
transform() (*ezdxf.math.Bezier4P method*), 451
transform() (*ezdxf.math.BSpline method*), 448

transform() (*ezdxf.math.ConstructionEllipse method*), 444
 transform() (*ezdxf.math.Matrix44 method*), 427
 transform() (*ezdxf.math.ucs method*), 423
 transform() (*ezdxf.path.Path method*), 474
 transform() (*ezdxf.render.MeshTransformer method*), 535
 transform_2d() (*ezdxf.tools.text.TextLine static method*), 496
 transform_direction() (*ezdxf.math.Matrix44 method*), 427
 transform_directions() (*ezdxf.math.Matrix44 method*), 427
 transform_paths() (*in module ezdxf.path*), 469
 transform_paths_to_ocs() (*in module ezdxf.path*), 469
 transform_vertices() (*ezdxf.math.Matrix44 method*), 427
 transformation_matrix_extruded_entity (*ezdxf.entities.ExtrudedSurface attribute*), 358
 transformation_matrix_path_entity() (*ezdxf.entities.SweptSurface method*), 360
 transformation_matrix_revolved_entity (*ezdxf.entities.RevolvedSurface attribute*), 360
 transformation_matrix_sweep_entity (*ezdxf.entities.SweptSurface attribute*), 360
 translate() (*ezdxf.entities.Circle method*), 291
 translate() (*ezdxf.entities.DXFGraphic method*), 285
 translate() (*ezdxf.entities.Ellipse method*), 301
 translate() (*ezdxf.entities.Insert method*), 245
 translate() (*ezdxf.entities.Line method*), 318
 translate() (*ezdxf.entities.Point method*), 343
 translate() (*ezdxf.entities.Ray method*), 351
 translate() (*ezdxf.entities.Text method*), 363
 translate() (*ezdxf.entities.XLine method*), 373
 translate() (*ezdxf.math.ConstructionArc method*), 439
 translate() (*ezdxf.math.ConstructionBox method*), 445
 translate() (*ezdxf.math.ConstructionCircle method*), 437
 translate() (*ezdxf.math.ConstructionLine method*), 436
 translate() (*ezdxf.math.Matrix44 class method*), 425
 translate() (*ezdxf.math.Shape2d method*), 447
 translate() (*ezdxf.render.MeshTransformer method*), 535
 transparency (*ezdxf.entities.DXFGraphic attribute*), 284
 transparency (*ezdxf.entities.DXFGraphic.dxf attribute*), 287
 transparency (*ezdxf.entities.Layer attribute*), 222
 transparency (*ezdxf.entities.MText.dxf attribute*), 332
 transparency2float() (*in module ezdxf.colors*), 402
 transpose() (*ezdxf.math.Matrix method*), 459
 transpose() (*ezdxf.math.Matrix44 method*), 427
 trashcan() (*ezdxf.entitydb.EntityDB method*), 687
 tridiagonal_matrix_solver() (*in module ezdxf.math*), 456
 tridiagonal_vector_solver() (*in module ezdxf.math*), 456
 true_color, 700
 true_color (*ezdxf.entities.DXFGraphic.dxf attribute*), 286
 true_color (*ezdxf.entities.Layer.dxf attribute*), 221
 true_color (*ezdxf.entities.Sun.dxf attribute*), 385
 ttf (*ezdxf.tools.fonts.FontFace attribute*), 501
 tuple() (*ezdxf.math.Vec3 class method*), 429
 tuples_to_tags() (*in module ezdxf.llodxf.types*), 689
 twist_angle (*ezdxf.entities.ExtrudedSurface.dxf attribute*), 358
 twist_angle (*ezdxf.entities.RevolvedSurface.dxf attribute*), 359
 twist_angle (*ezdxf.entities.SweptSurface.dxf attribute*), 360
 type (*ezdxf.entities.ArcEdge attribute*), 311
 type (*ezdxf.entities.EdgePath attribute*), 309
 type (*ezdxf.entities.EllipseEdge attribute*), 311
 type (*ezdxf.entities.LineEdge attribute*), 311
 type (*ezdxf.entities.PolylinePath attribute*), 308
 type (*ezdxf.entities.SplineEdge attribute*), 312

U

u_count (*ezdxf.entities.Surface.dxf attribute*), 357
 u_pixel (*ezdxf.entities.Image.dxf attribute*), 314
 UCS (*class in ezdxf.math*), 422
 ucs (*ezdxf.document.Drawing attribute*), 201
 ucs (*ezdxf.entities.View.dxf attribute*), 237
 ucs (*ezdxf.sections.tables.TablesSection attribute*), 211
 ucs() (*ezdxf.entities.Insert method*), 246
 ucs() (*ezdxf.entities.MText method*), 333
 ucs() (*ezdxf.entities.UCSTableEntry method*), 239
 ucs() (*ezdxf.math.Matrix44 static method*), 426
 ucs_base_handle (*ezdxf.entities.Viewport.dxf attribute*), 371
 ucs_handle (*ezdxf.entities.View.dxf attribute*), 238
 ucs_handle (*ezdxf.entities.Viewport.dxf attribute*), 371
 ucs_icon (*ezdxf.entities.Viewport.dxf attribute*), 371
 ucs_icon (*ezdxf.entities.VPort.dxf attribute*), 236
 ucs_origin (*ezdxf.entities.View.dxf attribute*), 237
 ucs_origin (*ezdxf.entities.Viewport.dxf attribute*), 371
 ucs_ortho_type (*ezdxf.entities.View.dxf attribute*), 237
 ucs_ortho_type (*ezdxf.entities.Viewport.dxf attribute*), 371

ucs_per_viewport (*ezdxf.entities.Viewport.dxf attribute*), 371
ucs_x_axis (*ezdxf.entities.Viewport.dxf attribute*), 371
ucs_xaxis (*ezdxf.entities.View.dxf attribute*), 237
ucs_y_axis (*ezdxf.entities.Viewport.dxf attribute*), 371
ucs_yaxis (*ezdxf.entities.View.dxf attribute*), 237
UCSTable (*class in ezdxf.sections.table*), 219
UCSTableEntry (*class in ezdxf.entities*), 239
uid (*ezdxf.entities.Body.dxf attribute*), 290
Underlay (*class in ezdxf.entities*), 366
underlay_def_handle (*ezdxf.entities.Underlay.dxf attribute*), 366
UnderlayDefinition (*class in ezdxf.entities*), 386
UNDERLINE (*ezdxf.enums.MTextStroke attribute*), 399
underline () (*ezdxf.tools.text.MTextEditor method*), 493
uniform_knot_vector () (*in module ezdxf.math*), 407
union () (*ezdxf.addons.pyccsg.CSG method*), 573
union () (*ezdxf.math.BoundingBox method*), 433
union () (*ezdxf.math.BoundingBox2d method*), 434
unit_circle () (*in module ezdxf.path*), 471
unit_name () (*in module ezdxf.units*), 38
unit_vector (*ezdxf.entities.Ray.dxf attribute*), 351
unit_vector (*ezdxf.entities.XLine.dxf attribute*), 373
Unitless (*ezdxf.enums.InsertUnits attribute*), 400
units (*ezdxf.document.Drawing attribute*), 201
units (*ezdxf.entities.BlockRecord.dxf attribute*), 240
units (*ezdxf.layouts.BaseLayout attribute*), 251
unlink_entity () (*ezdxf.layouts.BaseLayout method*), 251
unlink_from_layout ()
 (*ezdxf.entities.DXFGraphic method*), 284
unlock () (*ezdxf.entities.Layer method*), 222
unnamed (*ezdxf.entities.dxfgroups.DXFGroup.dxf attribute*), 279
up_direction (*ezdxf.entities.GeoData.dxf attribute*), 378
update () (*ezdxf.entities.DimStyleOverride method*), 296
update () (*ezdxf.lldxftags.Tags method*), 691
update_all () (*ezdxf.entities.MLineStyle method*), 325
update_dxf_attribs () (*ezdxf.entities.DXFEntity method*), 282
update_geometry () (*ezdxf.entities.MLine method*), 324
update_instance_counters ()
 (*ezdxf.sections.classes.ClassesSection method*), 210
update_paper () (*ezdxf.layouts.Layout method*), 276
upper (*ezdxf.math.BandedMatrixLU attribute*), 460
upper_right (*ezdxf.entities.VPort.dxf attribute*), 235
upright () (*in module ezdxf.upright*), 481
upright_all () (*in module ezdxf.upright*), 481
upright_text_angle ()
 (*in module ezdxf.tools.text*), 498
use_auto_height (*ezdxf.entities.MTextData attribute*), 342
use_c_ext (*in module ezdxf.options*), 508
use_matplotlib (*in module ezdxf.options*), 509
use_plot_styles () (*ezdxf.layouts.Layout method*), 276
use_standard_scale ()
 (*ezdxf.layouts.Layout method*), 276
use_window_bg_color (*ezdxf.entities.MTextData attribute*), 342
use_word_break (*ezdxf.entities.MTextData attribute*), 342
used_dimstyles (*ezdxf.addons.importer.Importer attribute*), 551
used_layers (*ezdxf.addons.importer.Importer attribute*), 551
used_linetypes (*ezdxf.addons.importer.Importer attribute*), 551
used_styles (*ezdxf.addons.importer.Importer attribute*), 551
user_location_override ()
 (*ezdxf.entities.DimStyleOverride method*), 299
user_scale_factor (*ezdxf.entities.GeoData.dxf attribute*), 378
UserRecord (*class in ezdxf.urecord*), 487
USSurveyFeet (*ezdxf.enums.InsertUnits attribute*), 400
USSurveyInch (*ezdxf.enums.InsertUnits attribute*), 400
USSurveyMile (*ezdxf.enums.InsertUnits attribute*), 400
USSurveyYard (*ezdxf.enums.InsertUnits attribute*), 400
ux (*ezdxf.math.OCS attribute*), 421
ux (*ezdxf.math.UCS attribute*), 422
uy (*ezdxf.math.OCS attribute*), 421
uy (*ezdxf.math.UCS attribute*), 422
uz (*ezdxf.math.OCS attribute*), 421
uz (*ezdxf.math.UCS attribute*), 422

V

v_count (*ezdxf.entities.Surface.dxf attribute*), 357
v_pixel (*ezdxf.entities.Image.dxf attribute*), 314
validate () (*ezdxf.document.Drawing method*), 204
valign (*ezdxf.entities.Text.dxf attribute*), 362
value (*ezdxf.entities.DictionaryVar.dxf attribute*), 376
value (*ezdxf.lldxftypes.DXFTag attribute*), 689
values (*ezdxf.lldxftags.TagArray attribute*), 694
values (*ezdxf.lldxftags.TagList attribute*), 694
values () (*ezdxf.entitydb.EntityDB method*), 687

varnames () (*ezdxf.sections.header.HeaderSection method*), 209
Vec2 (*class in ezdxf.math*), 431
vec2 (*ezdxf.math.Vec3 attribute*), 428
Vec3 (*class in ezdxf.math*), 427
vector (*ezdxf.math.Plane attribute*), 432
version (*ezdxf.entities.Body.dxf attribute*), 290
version (*ezdxf.entities.Dimension.dxf attribute*), 292
version (*ezdxf.entities.GeoData.dxf attribute*), 377
version (*ezdxf.entities.Mesh.dxf attribute*), 326
version (*ezdxf.entities.MultiLeader.dxf attribute*), 338
version (*ezdxf.entities.Sun.dxf attribute*), 385
Vertex (*class in ezdxf.entities*), 348
vertex_at () (*ezdxf.math.ConstructionPolyline method*), 446
VERTEX_SIZE (*ezdxf.lldxf.packedtags.VertexArray attribute*), 694
VertexArray (*class in ezdxf.lldxf.packedtags*), 694
vertical_unit_scale (*ezdxf.entities.GeoData.dxf attribute*), 378
vertical_units (*ezdxf.entities.GeoData.dxf attribute*), 378
vertices (*ezdxf.entities.Leader attribute*), 317
vertices (*ezdxf.entities.LeaderLine attribute*), 340
vertices (*ezdxf.entities.Mesh attribute*), 326
vertices (*ezdxf.entities.MeshData attribute*), 326
vertices (*ezdxf.entities.MeshVertexCache attribute*), 349
vertices (*ezdxf.entities.MLine attribute*), 323
vertices (*ezdxf.entities.Polyline attribute*), 345
vertices (*ezdxf.entities.PolylinePath attribute*), 308
vertices (*ezdxf.math.Shape2d attribute*), 447
vertices (*ezdxf.render.MeshBuilder attribute*), 533
vertices () (*ezdxf.disassemble.Primitive method*), 476
vertices () (*ezdxf.entities.Circle method*), 291
vertices () (*ezdxf.entities.Ellipse method*), 301
vertices () (*ezdxf.entities.LWPolyline method*), 320
vertices () (*ezdxf.entities.Solid method*), 354
vertices () (*ezdxf.entities.Trace method*), 365
vertices () (*ezdxf.math.ConstructionArc method*), 439
vertices () (*ezdxf.math.ConstructionCircle method*), 437
vertices () (*ezdxf.math.ConstructionEllipse method*), 443
vertices_in_wcs () (*ezdxf.entities.LWPolyline method*), 321
View (*class in ezdxf.entities*), 236
view_brightness (*ezdxf.entities.Viewport.dxf attribute*), 372
view_center_point (*ezdxf.entities.Viewport.dxf attribute*), 369
view_contrast (*ezdxf.entities.Viewport.dxf attribute*), 372
view_direction_vector (*ezdxf.entities.Viewport.dxf attribute*), 369
view_height (*ezdxf.entities.Viewport.dxf attribute*), 369
view_mode (*ezdxf.entities.View.dxf attribute*), 237
view_mode (*ezdxf.entities.VPort.dxf attribute*), 236
view_target_point (*ezdxf.entities.Viewport.dxf attribute*), 369
view_twist (*ezdxf.entities.View.dxf attribute*), 237
view_twist (*ezdxf.entities.VPort.dxf attribute*), 236
view_twist_angle (*ezdxf.entities.Viewport.dxf attribute*), 369
Viewport (*class in ezdxf.entities*), 368
viewports (*ezdxf.document.Drawing attribute*), 200
viewports (*ezdxf.sections.tables.TablesSection attribute*), 212
viewports () (*ezdxf.layouts.Paperspace method*), 277
ViewportTable (*class in ezdxf.sections.table*), 220
views (*ezdxf.document.Drawing attribute*), 200
views (*ezdxf.sections.tables.TablesSection attribute*), 212
ViewTable (*class in ezdxf.sections.table*), 219
virtual_entities () (*ezdxf.entities.Dimension method*), 295
virtual_entities () (*ezdxf.entities.Insert method*), 245
virtual_entities () (*ezdxf.entities.Leader method*), 317
virtual_entities () (*ezdxf.entities.LWPolyline method*), 322
virtual_entities () (*ezdxf.entities.MLine method*), 324
virtual_entities () (*ezdxf.entities.MultiLeader method*), 338
virtual_entities () (*ezdxf.entities.Point method*), 343
virtual_entities () (*ezdxf.entities.Polyline method*), 347
virtual_entities () (*ezdxf.render.trace.CurvedTrace method*), 539
virtual_entities () (*ezdxf.render.trace.LinearTrace method*), 538
virtual_entities () (*ezdxf.render.trace.TraceBuilder method*), 537
virtual_entities () (*in ezdxf.render.point*), 539
virtual_guide (*ezdxf.entities.LoftedSurface.dxf attribute*), 359
virtual_mtext_entity () (*ezdxf.entities.AttDef*)

method), 248
virtual_mtext_entity() (ezdxf.entities.Attrib method), 247
virtual_pen_number (ezdxf.addons.acadctb.PlotStyle attribute), 577
visual_style_handle (ezdxf.entities.View.dxf attribute), 238
visual_style_handle (ezdxf.entities.Viewport.dxf attribute), 371
VPort (class in ezdxf.entities), 235
vtx0 (ezdxf.entities.Face3d.dxf attribute), 287
vtx0 (ezdxf.entities.Solid.dxf attribute), 354
vtx0 (ezdxf.entities.Trace.dxf attribute), 365
vtx1 (ezdxf.entities.Face3d.dxf attribute), 287
vtx1 (ezdxf.entities.Solid.dxf attribute), 354
vtx1 (ezdxf.entities.Trace.dxf attribute), 365
vtx1 (ezdxf.entities.Vertex.dxf attribute), 348
vtx2 (ezdxf.entities.Face3d.dxf attribute), 287
vtx2 (ezdxf.entities.Solid.dxf attribute), 354
vtx2 (ezdxf.entities.Trace.dxf attribute), 365
vtx2 (ezdxf.entities.Vertex.dxf attribute), 348
vtx3 (ezdxf.entities.Face3d.dxf attribute), 287
vtx3 (ezdxf.entities.Solid.dxf attribute), 354
vtx3 (ezdxf.entities.Trace.dxf attribute), 365
vtx3 (ezdxf.entities.Vertex.dxf attribute), 348
vtx4 (ezdxf.entities.Vertex.dxf attribute), 348

W

was_a_proxy (ezdxf.entities.DXFClass.dxf attribute), 211
wcs_to_crs() (ezdxf.addons.geo.GeoProxy method), 548
wcs_vertices() (ezdxf.entities.Face3d method), 288
wcs_vertices() (ezdxf.entities.Solid method), 354
wcs_vertices() (ezdxf.entities.Trace method), 366
wedge() (in module ezdxf.path), 471
weight (ezdxf.tools.fonts.FontFace attribute), 502
weights (ezdxf.entities.Spline attribute), 356
weights (ezdxf.entities.SplineEdge attribute), 312
weights() (ezdxf.math.BSpline method), 448
wgs84_3395_to_4326() (in module ezdxf.addons.geo), 549
wgs84_4326_to_3395() (in module ezdxf.addons.geo), 549
WHITE (ezdxf.enums.ACI attribute), 402
width (ezdxf.entities.AttribData attribute), 341
width (ezdxf.entities.MText.dxf attribute), 331
width (ezdxf.entities.MTextData attribute), 341
width (ezdxf.entities.Text.dxf attribute), 361
width (ezdxf.entities.Textstyle.dxf attribute), 225
width (ezdxf.entities.View.dxf attribute), 237
width (ezdxf.entities.Viewport.dxf attribute), 368
width (ezdxf.math.ConstructionBox attribute), 444
width (ezdxf.tools.text_size.ezdxf.tools.text_size.TextSize attribute), 499
width_factor() (ezdxf.tools.text.MTextEditor method), 493
win_exec_path (in module ezdxf.addons.odafc), 565
WINDOW (ezdxf.enums.MTextBackgroundColor attribute), 399
window() (in module ezdxf.zoom), 510
Wipeout (class in ezdxf.entities), 372
with_changes() (ezdxf.addons.drawing.config.Configuration method), 542
write() (ezdxf.addons.acadctb.ColorDependentPlotStyles method), 576
write() (ezdxf.addons.acadctb.NamedPlotStyles method), 577
write() (ezdxf.addons.itedxf.IterDXFWriter method), 558
write() (ezdxf.document.Drawing method), 201
write() (in module ezdxf.options), 505
write_file() (in module ezdxf.options), 505
write_fixed_meta_data_for_testing (in module ezdxf.options), 508
write_home_config() (in module ezdxf.options), 505
write_to_header() (ezdxf.gfxattribs.GfxAttribs method), 492

X

x (ezdxf.math.Vec3 attribute), 428
X_AXIS (in module ezdxf.math), 431
x_height (ezdxf.tools.fonts.FontMeasurements attribute), 502
x_rotate() (ezdxf.math.Matrix44 class method), 425
xaxis (ezdxf.entities.UCSTableEntry.dxf attribute), 239
XData (class in ezdxf.entities.xdata), 695
xdata (ezdxf.entities.xdata.XDataUserDict attribute), 485
xdata (ezdxf.entities.xdata.XDataUserList attribute), 484
xdata (ezdxf.lldx_extendedtags.ExtendedTags attribute), 692
XDataUserDict (class in ezdxf.entities.xdata), 485
XDataUserList (class in ezdxf.entities.xdata), 484
XLine (class in ezdxf.entities), 373
xof() (ezdxf.math.ConstructionRay method), 435
XRecord (class in ezdxf.entities), 387
xrecord (ezdxf.urecord.BinaryRecord attribute), 488
xrecord (ezdxf.urecord.UserRecord attribute), 487
xref_path (ezdxf.entities.Block.dxf attribute), 242
xround() (in module ezdxf.math), 408
xscale (ezdxf.entities.Insert.dxf attribute), 243
xscale (ezdxf.entities.Shape.dxf attribute), 352
xy (ezdxf.math.Vec3 attribute), 428
xyz (ezdxf.math.Vec3 attribute), 428

`xyz_rotate()` (*ezdxf.math.Matrix44 class method*),
425

Y

`y` (*ezdxf.math.Vec3 attribute*), 428
`Y_AXIS` (*in module ezdxf.math*), 431
`y_rotate()` (*ezdxf.math.Matrix44 class method*), 425
`Yards` (*ezdxf.enums.InsertUnits attribute*), 400
`yaxis` (*ezdxf.entities.UCSTableEntry.dxf attribute*), 239
`YELLOW` (*ezdxf.enums.ACI attribute*), 402
`yof()` (*ezdxf.math.ConstructionRay method*), 435
`yscale` (*ezdxf.entities.Insert.dxf attribute*), 243

Z

`z` (*ezdxf.math.Vec3 attribute*), 428
`Z_AXIS` (*in module ezdxf.math*), 431
`z_rotate()` (*ezdxf.math.Matrix44 class method*), 425
`zoom_to_paper_on_update()`
 (*ezdxf.layouts.Layout method*), 276
`zscale` (*ezdxf.entities.Insert.dxf attribute*), 243