pyGenCAD Documentation

Release 0.2.1

Ed Blake <kitsu.eb@gmail.com>

Apr 04, 2017

Contents

Description Python module for generating CAD software command scripts.

Author Ed Blake <kitsu.eb@gmail.com>

Date Jun. 19 2013

CHAPTER 1

Changes

Change 9 Added features to svg module, got typsec example working for svg. (Ed Blake Sun Jun 02 10:06:47 2013 -0700)

Change 8 SVG refactor to ElementTree complete. (Ed Blake Thu May 30 12:28:40 2013 -0700)

Change 7 Started typsec demo, added svg module with all methods and tests written. (Ed Blake Fri May 24 16:23:01 2013 -0700)

Change 6 Added notes on script output usage (with links). (Ed Blake Sun May 19 18:34:19 2013 -0700)

Change 5 Repo public, docs online, published to PyPI. (Ed Blake Sun May 19 17:38:06 2013 -0700)

CHAPTER 2

Introduction

A pidgin, or pidgin language, is a simplified language that develops as a means of communication between two or more groups that do not have a language in common. It is most commonly employed in situations such as trade, or where both groups speak languages different from the language of the country in which they reside (but where there is no common language between the groups). (Via Wikipedia)

This module provides a simple interface to generate command scripts targeting various CAD software via Python. Low level methods are provided for inputting raw commands, single and multiple coordinate input, and setting the active layer/level. Convenience methods are also provided for drawing lines, polylines, circles, and text. Methods are also provided for storing element references and moving, copying, rotating, scaling, and erasing elements by reference.

A uniform interface is provided across all supported backends allowing one Python script to output to multiple platforms using identical code. Additional backend specific features may be available, and arbitrary code can be generated using the 'cmd' method.

CHAPTER 3

Requirements

This is a pure Python module with no external dependencies (except a supported backend to execute your output).

The module was written and tested against Python2.7 and is using several non-backward compatible features:

- Non-indexed string format placeholders *{}*
- Chained context managers
- Extended unittest asserts

Adding Python >= 3.0 support should be trivial, but I am not currently using Python 3 anywhere.

CHAPTER 4

Installation

Download the repo archive, unzip, and run *python setup.py install* in the unzipped directory.

Alternately use pip to get the latest version uploaded to PyPI *pip install pygencad*.

CHAPTER 5

Testing

A helper script "runtests.py" is provided in the project root to run the entire test suite. Runnable test scripts are provided for each sub-module in the test folder.

Additionally the cover_tests.py script will run all the tests using coverage.py, with branch coverage, and open the html report.

Also running build_docs.py will run doctests as part of the sphinx build.

CHAPTER 6

Usage

Importing the pygencad module brings in a named module for each backend, a dict of name:backend pairs and name:ext pairs, and two helper functions. In normal usage you would call the get_script function with a filelike and the name of your desired backend to get started:

```
import pygencad as pgc

# Notice you are responsible for the life of your filelike
filename = 'outfile' + pgc.backext['autocad']
with open(filename, 'w') as outfile:
    # Script objects handle context setup and teardown
    with pgc.get_script(outfile, 'autocad') as script:
        script.cmd("pass")
```

The other helper function returns a new layer object for the current backend:

```
my_layer = pgc.get_layer('autocad')('mylayer', co=1)
```

Instantiated script objects also have a reference to their Layer class:

```
my_layer = script.Layer('mylayer', co=1)
```

The script object also provides a layer context manager:

```
with script.layer(my_layer):
    # Draw circle on my layer
    script.circle((0,0), 5)
# The layer method creates a new layer if passed layer params
with script.layer('extra', co=2):
    script.line((-2.5,-2.5), (2.5, 2.5))
```

After you're script has run you should get a script you can run with you're backend of choice. AutoCAD scripts are run by using the *script* command, or by dragging the script file onto the AutoCAD window. MicroStation scripts are run by typing @ followed by the DOS short name path to you're script in the keyin editor. SVG output is just bare <svg> tags, Wrapping in a minimal HTML document is recomended.

See the specific backend modules in the docs for more info.

CHAPTER 7

ToDo

- Add support for setting element overrides globally and per method call.
- Write cool examples, both general and backend specific.
- Add a mirror modification method and/or implement non-uniform scaling.
- Add AutoCAD point mode support.
- Add some more AutoLisp utility code to the autocad module.
- Move block/cell code to base class (name?). (SVG use defs and links as blocks)
- Improve MicroStation specific methods.
- **Add support for more backends:**
  - Blender
  - VPython
  - ???

CHAPTER 8

Sections

commands

Main PyGenCad entry point.

The commands sub-module provides helper functions for creating CommandFile and Layer objects targeting specific backends. Functions from the commands sub-module are available at the top level of the PyGenCAD package.

```
>>> from pygencad import *
>>> from StringIO import StringIO
>>> f = StringIO()
>>> with get_script(f, 'autocad') as script:
...     script.cmd("Hello, World!")
...
'1'
>>> "Hello, World!" in f.getvalue()
True
```

variables

pygencad.**backends**

Mapping of backend names to backend modules.

---

pygencad.**backext**

Mapping of backend names to backend file extensions.

---

functions

pygencad.**get_script**(*filelike*, *backend*)

Return an initialized CommandFile object for the requested backend.

```
>>> from pygencad import *
>>> from StringIO import StringIO
>>> f = StringIO()
>>> get_script(f, 'autocad')
<pygencad.autocad.CommandFile object at 0x...>
>>> get_script(f, 'spam')
Traceback (most recent call last):
    ...
ValueError: Unknown backend ...!
```

Parameters

- **filelike** (*filelike*) – An object with a write method.
- **backend** (*string*) – Name of the backend to use, must be in pygencad.backends.

Raises ValueError if the provided name isn't in pygencad.backends

Returns An instantiated CommandFile object for the indicated backend.

---

pygencad.**get_layer**(*backend*)

Return the layer class for the indicated backend.

```
>>> from pygencad import *
>>> from StringIO import StringIO
>>> f = StringIO()
>>> get_layer('autocad') # Notice the return isn't an object:
<class 'pygencad.autocad.Layer'>
>>> get_script(f, 'spam')
Traceback (most recent call last):
    ...
ValueError: Unknown backend ...!
```

Parameters **backend** (*string*) – Name of the backend to use, must be in pygencad.backends.

Raises ValueError if the provided name isn't in pygencad.backends

Returns Layer class for the indicated backend.

commandfile

Base classes for implementing drawing backend wrappers.

The base CommandFile class methods implement AutoCAD like outputs, and some are used un-modified in the autocad.CommandFile implementation.

classes

Layer

class pygencad.commandfile.**Layer**(*name*, *co=None*, *lw=None*, *lc=None*)

Object representing a grouping and styling context.

Layer like objects are expected to support at least naming, color, line weight, and line type. All properties except name are expected to be optional.

```
>>> from pygencad.commandfile import Layer
>>> l = Layer("test")
>>> l
<pygencad.commandfile.Layer object at 0x...>
>>> print l
Layer:
    lv=test
    co=None
    lw=None
    lc=None
```

Parameters

- **name** (*string OR Layer-like*) – The name visible in the backend after the script is run. A tuple of layer properties, or a Layer object may be passed as the first argument (name) in which case the new layer is initialized with provided values.
- **co** – The color of the layer.
- **lw** – The line-weight of the layer.
- **lc** – The line-class (type) of the layer.

Store properties and setup template.

__str__()
Render this layer as a string.

The __str__ method is used to output the commands necessary to set the Layer objects properties active in the script context.

CommandFile

class pygencad.commandfile.**CommandFile**(*filelike*, *setup=None*, *teardown=None*)
Script generation interface.

Provides convenience methods for issuing common commands, managing script context, and for issuing arbitrary commands.

```
>>> from pygencad.commandfile import CommandFile
>>> from StringIO import StringIO
>>> f = StringIO()
>>> with CommandFile(f) as script:
...     script.cmd("test")
...
'1'
>>> print f.getvalue()
test
```

Parameters

- **filelike** (*filelike*) – An object with a write method.
- **setup** (*string OR callable*) – Commands to include at the beginning of the script.
- **teardown** (*string OR callable*) – Commands to include at the end of the script.

class Layer(*name*, *co=None*, *lw=None*, *lc=None*)
A local binding to the backend specific Layer class

CommandFile.**setup**()
Write default configuration info and run user provided setup func.

Called automatically by CommandFile context manager. Writes default setup code, then adds any user provided setup. If the user setup is callable the return value is written to the script, otherwise the user setup is written as a string.

CommandFile.**teardown**()
Write cleanup commands and run user provided teardown func.

Called automatically by CommandFile context manager. Writesany user provided teardown, then writes default teardown code. If the user teardown is callable the return value is written to the script, otherwise the user teardown is written as a string.

CommandFile.**layer**(**args*, ***kwds*)
Script layer context manager.

Activates the provided layer settings, and resets layer state on exit.

```
>>> from pygencad.commandfile import CommandFile
>>> from StringIO import StringIO
>>> f = StringIO()
>>> with CommandFile(f) as script:
...     with script.layer('layer'):
...         script.cmd("test")
...
'1'
>>> print f.getvalue()
Layer:
    lv=layer
    co=None
    lw=None
    lc=None
test
```

Parameters

- ***args** – Valid Layer arguments.
- ****kwargs** – Valid Layer arguments.

CommandFile.**set_layer**(*layer*, **args*, ***kwargs*)
Activates the provided layer object or description.

Parameters

- **layer** – A layer object, or a valid Layer name argument.
- ***args** – Valid Layer arguments.
- ****kwargs** – Valid Layer arguments.

CommandFile.**pop_layer**()
Restores active layer prior to last set_layer call.

CommandFile.**cmd**(*command*, **args*)
Write an arbitrary script command.

If the command string contains format expressions, and additional arguments were passed, the command string will be formated using the additional arguments.

```
>>> from pygencad.commandfile import CommandFile
>>> from StringIO import StringIO
>>> f = StringIO()
>>> with CommandFile(f) as script:
...     script.cmd("command with data: {:0.5f}", 355/113.0)
...
'1'
>>> print f.getvalue()
command with data: 3.14159
```

Parameters

- **command** (*string*) – The command to be written to the script.

- ***args** – Values to be formated by the command string.

Returns Some reference to any object created.

CommandFile.**point**(*p*, *reset=False*, *mode=None*, *write=True*)

Convert an iterable into a point of the correct format.

This method should be overridden in sub-classes.

```
>>> from pygencad.commandfile import CommandFile
>>> from StringIO import StringIO
>>> f = StringIO()
>>> point = (1, 2, 3)
>>> with CommandFile(f) as script:
...     script.point(point)
...     script.point(point, reset=True)
...     script.point(point, mode='special')
...     p = script.point(point, write=False)
...     script.cmd("Here is the point: {}", p)
...
'1,2,3'
'1,2,3\n'
'1,2,3'
'1'
>>> print f.getvalue()
1,2,3
1,2,3

1,2,3
Here is the point: 1,2,3
```

Parameters

- **p** – The point to be converted.
- **reset** (*boolean*) – Optionally append the result of calling the reset method after writing the point.
- **mode** (*string*) – Optional point output mode (unimplemented in base class).
- **write** (*boolean*) – Flag for writing output to script file.

Returns The point converted to a string.

CommandFile.**points**(*ps*, *reset=False*, *mode=None*)

Calls CommandFile.point on each member of an iterable.

This method may need to be overridden in sub-classes.

```
>>> from pygencad.commandfile import CommandFile
>>> from StringIO import StringIO
>>> f = StringIO()
>>> with CommandFile(f) as script:
...     script.points((
...         (1, 2),
...         (3, 4),
...         (5, 6),
...     ))
...
>>> print f.getvalue()
```

```
1,2
3,4
5,6
```

Parameters

- **ps** – The points to be converted.
- **reset** (*boolean*) – Optionally append the result of calling the reset method after writing all points.
- **mode** (*string*) – Optional point output mode (unimplemented in base class).

CommandFile.**reset**(*write=True*)

Write the sequence required to exit the current command.

This method may need to be overridden in sub-classes.

Parameters **write** (*boolean*) – Flag for writing output to script file.

Returns The command that output by reset as a string.

CommandFile.**line**(*ps*, *reset=False*, *mode=None*)

Write a line segment command for each pair of points provided

This method should be overridden in sub-classes. The output command for the line method is expected to produce a set of disconnected line segments. See the polyline command for outputting connected "line strings".

Parameters

- **ps** – The end points of the lines.
- **reset** (*boolean*) – Optionally append the result of calling the reset method after writing all points.
- **mode** (*string*) – Optional point output mode unimplemented in base class.

Returns Some reference to any object created.

CommandFile.**polyline**(*ps*, *close=True*, *reset=False*, *mode=None*)

Write a string of lines connected at each of the provided points.

This method should be overridden in sub-classes. The output command for the polyline method is expected to produce a single object connecting all the provided points in order. The object produced is expected to be "closable". Otherwise the method should ensure that the last point is equal to the first to close the polyline.

Parameters

- **ps** – The ordered points forming the line.
- **close** (*boolean*) – Flag indicating whether the line should end where it started.
- **reset** (*boolean*) – Optionally append the result of calling the reset method after writing all points.
- **mode** (*string*) – Optional point output mode unimplemented in base class.

Returns Some reference to any object created.

CommandFile.**circle**(*center*, *radius*)

Write a circle command.

This method should be overridden in sub-classes.

Parameters

- **center** (*iterable*) – Point in a format passable to CommandFile.point.
- **radius** (*number*) – Radius of the circle.

Returns Some reference to any object created.

CommandFile.**text**(*text*, *p*, *height=1.0*)

Write a text placement command.

This method should be overridden in sub-classes.

Parameters

- **text** (*string*) – The text to place.
- **p** (*iterable*) – The insertion point in a format passable to CommandFile.point.
- **height** (*number*) – How tall the text lines are.

Returns Some reference to any object created.

CommandFile.**store**(*name*, **els*)

Store provided elements to be manipulated later.

This method should be overridden in sub-classes.

This method provides a uniform interface for grouping objects regardless of backend. This is required for the move/copy/rotate/scale methods to be of any practical use. Each backend's store method should transparently handle combining elements and groups.

Parameters

- **name** (*backend specific*) – The id used to store the selection.
- ***els** (*backend specific*) – The elements to store.

Raises ValueError if no elements are provided.

Returns An element group passable to move/copy/rotate/scale methods.

CommandFile.**move**(*els*, *base=(0, 0)*, *dest=(0, 0)*)

Transform the provided elements from base to dest.

Parameters

- **els** (*backend specific*) – The elements to move.
- **base** (*iterable*) – The base point to transform from.
- **dest** (*iterable*) – The dest point to transform to.

Returns Some reference to modified objects.

CommandFile.**copy**(*els*, *base=(0, 0)*, *dest=(0, 0)*)

Transform duplicates of the provided elements from base to dest.

Parameters

- **els** (*backend specific*) – The elements to copy.
- **base** (*iterable*) – The base point to transform from.
- **dest** (*iterable*) – The dest point to transform to.

Returns Some reference to duplicated objects.

CommandFile.**rotate**(*els*, *base=(0, 0)*, *ang=0*)
Rotate the provided elements from base by angle.

Parameters

- **els** (*backend specific*) – The elements to rotate.
- **base** (*iterable*) – The base point to transform from.
- **ang** (*number*) – The angle to rotate in degrees (counter-clockwise).

Returns Some reference to modified objects.

CommandFile.**scale**(*els*, *base=(0, 0)*, *scale=1*)
Scale the provided elements from base by scale.

Remember that scaling an axis by -1 is equivalent to mirroring.

Parameters

- **els** (*backend specific*) – The elements to scale.
- **base** (*iterable*) – The base point to transform from.
- **scale** (*number*) – The scale factor to apply.

Returns Some reference to modified objects.

CommandFile.**erase**(*els*)
Remove the indicated elements.

Parameters **els** (*backend specific*) – The elements to remove.

autocad

AutoCAD expects scripts to have the '.scr' extension. The generated scripts can be run using the AutoCAD script command, or by dragging the script file onto the AutoCAD window. For more information see the AutoCAD help. Wrapper for AutoCAD command script generation.

classes

- *Layer*
- ***CommandFile***
  - *CommandFile.setup()*
  - *CommandFile.teardown()*
  - *CommandFile.ucs()*
  - *CommandFile.set_ucs()*
  - *CommandFile.pop_ucs()*
  - *CommandFile.point()*
  - *CommandFile.points()*
  - *CommandFile.reset()*
  - *CommandFile.line()*
  - *CommandFile.polyline()*

- *CommandFile.poly3d()*
- *CommandFile.circle()*
- *CommandFile.cmd()*
- *CommandFile.layer()*
- *CommandFile.set_layer()*
- *CommandFile.pop_layer()*
- *CommandFile.text()*
- *CommandFile.block()*
- *CommandFile.store()*
- *CommandFile.move()*
- *CommandFile.copy()*
- *CommandFile.rotate()*
- *CommandFile.scale()*
- *CommandFile.erase()*

Layer

class pygencad.autocad.**Layer**(*name*, *co=None*, *lw=None*, *lc=None*, *desc=''*, *st=''*, *plt=True*)

AutoCAD layer wrapper.

In addition to color, line-weight, and line-type you can also set plot-style, non-plot, and description. The co, lw, and lc values default to "bylevel" which means that the attribute is not output in the script.

Parameters

- **name** (*string OR Layer-like*) – The name visible in Acad after the script is run. A tuple of layer properties, or a Layer object may be passed as the first argument (name) in which case the new layer is initialized with provided values.
- **co** (*int*) – The color of the layer.
- **lw** (*number*) – The line-weight of the layer.
- **lc** (*string*) – The line-class (type) of the layer.
- **desc** (*string*) – The layer description.
- **st** (*string*) – The layer style.
- **plt** (*boolean*) – Whether the layer is can be plotted.

__str__()

Render this layer as a string.

The __str__ method is used to output the commands necessary to set the Layer objects properties active in AutoCAD.

CommandFile

class pygencad.autocad.**CommandFile**(*filelike*, *setup=None*, *teardown=None*)
Wrapper for AutoCAD script generation.

In addition to implementing the methods defined in the base class this class provides a UCS context manager, 3d polyline method, and block method.

Parameters

- **filelike** (*filelike*) – An object with a write method.
- **setup** (*string OR callable*) – Commands to include at the beginning of the script.
- **teardown** (*string OR callable*) – Commands to include at the end of the script.

class **Layer**(*name*, *co=None*, *lw=None*, *lc=None*, *desc=''*, *st=''*, *plt=True*)
Local binding to the Acad Layer class

CommandFile.**setup**()
Write default configuration info and run user provided setup func.

The default setup stores the active osnaps in a variable named "osmodeinit" and turns off osnaps (which normally interfere with scripts). The initial value of cmdecho is recorded as "cmdechoinit" and cmdecho is also disabled.

CommandFile.**teardown**()
Write cleanup commands and run user provided teardown func.

The initial values recorded in the setup method are restored.

CommandFile.**ucs**(**args*, ***kwds*)
User Coordinate System context manager.

Write a set of UCS commands and rollback changes when done.

```
>>> from pygencad.autocad import CommandFile
>>> from StringIO import StringIO
>>> f = StringIO()
>>> script = CommandFile(f)
>>> with script.ucs('ucs w\nucs zaxis', ((0,0,0), (1,1,1)), 2):
...     # Draw a circle at origin pointing to the positive quadrant
...     script.circle((0,0,0), 5)
...
'1'
>>> print f.getvalue()
ucs w
ucs zaxis
0,0,0
1,1,1
circle 0,0,0 5
ucs p
ucs p
```

Parameters

- **command** (*string*) – UCS commands to execute on entry.
- **points** (*iterable*) – A list of points required by the *last* UCS command.
- **count** (*number*) – How many UCS commands were issued.

CommandFile.**set_ucs**(*command*, *points=None*)
Change the active UCS.

Parameters

- **command** (*string*) – UCS commands to execute on entry.
- **points** (*iterable*) – A list of points required by the *last* UCS command.

CommandFile.**pop_ucs**(*count=1*)
Rollback the indicated number of UCS changes.

Parameters **count** (*number*) – How many UCS commands need to be rolled back.

CommandFile.**point**(*p*, *reset=False*, *mode=None*, *write=True*)
Convert an iterable into a point of the correct format.

The mode argument is currently unsupported, but should support relative, polar, and relative polar coordinate input.

Parameters

- **p** – The point to be converted.
- **reset** (*boolean*) – Optionally append the result of calling the reset method after writing the point.
- **mode** (*string*) – Optional point output mode.
- **write** (*boolean*) – Flag for writing output to script file.

Returns The point converted to a string.

CommandFile.**points**(*ps*, *reset=False*, *mode=None*)
Calls CommandFile.point on each member of an iterable.

The mode Argument is currently unsupported, but should support relative, polar, and relative polar coordinate input.

Parameters

- **ps** – The points to be converted.
- **reset** (*boolean*) – Optionally append the result of calling the reset method after writing all points.
- **mode** (*string*) – Optional point output mode.

CommandFile.**reset**(*write=True*)
Write the sequence required to exit the current command.

For Acad this method currently just outputs a newline.

Parameters **write** (*boolean*) – Flag for writing output to script file.

Returns The command that output by reset as a string.

CommandFile.**line**(*ps*, *reset=False*, *mode=None*)
Write the Acad line command.

The mode argument is currently unsupported.

Parameters

- **ps** – The end points of the lines.
- **reset** (*boolean*) – Optionally append the result of calling the reset method after writing all points.

- **mode** (*string*) – Optional point output mode.

Returns Reference to created object.

CommandFile.**polyline**(*ps*, *close=True*, *reset=False*, *mode=None*)
Write the Acad pline command.

Support could be added in the future for drawing arc segments or other special pline settings.

The mode argument is currently unsupported.

Parameters

- **ps** – The ordered points forming the line.
- **close** (*boolean*) – Flag indicating whether the line should end where it started.
- **reset** (*boolean*) – Optionally append the result of calling the reset method after writing all points.
- **mode** (*string*) – Optional point output mode.

Returns Reference to created object.

CommandFile.**poly3d**(*ps*, *close=True*, *reset=False*, *mode=None*)
Write the Acad 3dpolyline command.

In Acad polylines are strictly 2d in the xy plane of the active UCS. To draw non-planer polylines you must use the 3dpolyline. While 2d plines support all properties, arc segments, and even splines, 3dpolylines are always straight segments, and don't support line-types (as of Acad2012).

The mode argument is currently unsupported.

Parameters

- **ps** – The ordered points forming the line.
- **close** (*boolean*) – Flag indicating whether the line should end where it started.
- **reset** (*boolean*) – Optionally append the result of calling the reset method after writing all points.
- **mode** (*string*) – Optional point output mode.

Returns Reference to created object.

CommandFile.**circle**(*center*, *radius*)
Write the Acad circle command.

Parameters

- **center** (*iterable*) – Point in a format passable to CommandFile.point.
- **radius** (*number*) – Radius of the circle.

Returns Reference to created object.

CommandFile.**text**(*text*, *p*, *height=1.0*, *ang=0*, *just='tl'*)
Write a text placement command.

Text is placed as mtext. The text object width is set to 0 (infinite) meaning the text does not automatically wrap. Support for setting width, and setting the active text style, could be added in the future.

Parameters

- **text** (*string*) – The text to place.
- **p** (*iterable*) – The insertion point in a format passable to CommandFile.point.

- **height** (*number*) – How tall the text lines are.
- **ang** (*number*) – The angle of the text in degrees.
- **just** (*string*) – The justification of the text, a pair of [t, c, b][l, c, r].

Returns Reference to created object.

CommandFile.**block**(*name*, *loc*, *ang=0*, *scale=1*, *attrs=None*)

Write a block placement command.

Currently only basic insertion is supported. Setting non-uniform scale factors, setting text values for attribute driven blocks, and setting dynamic block attributes could be added in the future.

The attrs argument is currently unsupported.

Parameters

- **name** (*string*) – The name of the block in Acad that is to be inserted.
- **loc** (*iterable*) – The block insertion point.
- **ang** (*number*) – The rotation angle of the inserted block in degrees.
- **scale** (*number*) – The uniform scale factor of the inserted block.
- **attrs** (*mapping*) – Attributes to be set for the block after insertion.

Returns Reference to created object.

CommandFile.**store**(*name*, **els*)

Store provided elements to be manipulated later.

For AutoCAD named selection sets are created (AutoCAD has a hard limit of 128 named selection sets). The selection set name will have a '!' prepended, and can be passed to any command expecting a selection. If els contains 'l' than only one item (the last drawn) is added to the stored selection. Alternately, if els contains only selections set names, a new joined selection is created.

Parameters

- **name** (*string*) – The id used to store the selection.
- ***els** (*'l' for last, or selection set names.*) – The elements to store.

Raises ValueError if no elements are provided.

Returns The name of the selection set.

CommandFile.**move**(*els*, *base=(0, 0)*, *dest=(0, 0)*)

Transform the provided elements from base to dest.

Parameters

- **els** (*A valid AutoCAD selection string.*) – The elements to move.
- **base** (*iterable*) – The base point to transform from.
- **dest** (*iterable*) – The dest point to transform to.

Returns Reference to modified objects.

CommandFile.**copy**(*els*, *base=(0, 0)*, *dest=(0, 0)*)

Transform duplicates of the provided elements from base to dest.

Parameters

- **els** (*A valid AutoCAD selection string.*) – The elements to copy.
- **base** (*iterable*) – The base point to transform from.

- **dest** (*iterable*) – The dest point to transform to.

Returns Reference to modified objects.

CommandFile.**rotate**(*els*, *base=(0, 0)*, *ang=0*)
Rotate the provided elements from base by angle.

Parameters

- **els** (*A valid AutoCAD selection string.*) – The elements to rotate.
- **base** (*iterable*) – The base point to transform from.
- **ang** (*number*) – The angle to rotate in degrees (counter-clockwise).

Returns Reference to modified objects.

CommandFile.**cmd**(*command*, **args*)
Write an arbitrary script command.

If the command string contains format expressions, and additional arguments were passed, the command string will be formated using the additional arguments.

```
>>> from pygencad.commandfile import CommandFile
>>> from StringIO import StringIO
>>> f = StringIO()
>>> with CommandFile(f) as script:
...     script.cmd("command with data: {:0.5f}", 355/113.0)
...
'1'
>>> print f.getvalue()
command with data: 3.14159
```

Parameters

- **command** (*string*) – The command to be written to the script.
- ***args** – Values to be formated by the command string.

Returns Some reference to any object created.

CommandFile.**layer**(**args*, ***kwds*)
Script layer context manager.

Activates the provided layer settings, and resets layer state on exit.

```
>>> from pygencad.commandfile import CommandFile
>>> from StringIO import StringIO
>>> f = StringIO()
>>> with CommandFile(f) as script:
...     with script.layer('layer'):
...         script.cmd("test")
...
'1'
>>> print f.getvalue()
Layer:
    lv=layer
    co=None
    lw=None
    lc=None
test
```

Parameters

- ***args** – Valid Layer arguments.
- ****kwargs** – Valid Layer arguments.

CommandFile.**pop_layer**()
Restores active layer prior to last set_layer call.

CommandFile.**scale**(*els*, *base=(0, 0)*, *scale=1*)
Scale the provided elements from base by scale.

Remember that scaling an axis by -1 is equivalent to mirroring.

Parameters

- **els** (*A valid AutoCAD selection string.*) – The elements to scale.
- **base** (*iterable*) – The base point to transform from.
- **scale** (*number*) – The scale factor to apply.

Returns Reference to modified objects.

CommandFile.**set_layer**(*layer*, **args*, ***kwargs*)
Activates the provided layer object or description.

Parameters

- **layer** – A layer object, or a valid Layer name argument.
- ***args** – Valid Layer arguments.
- ****kwargs** – Valid Layer arguments.

CommandFile.**erase**(*els*)
Remove the indicated elements.

Parameters **els** (*A valid AutoCAD selection string.*) – The elements to remove.

microstation

MicroStation expects scripts with the '.txt' extension. Scripts can be run by typing @ followed by the DOS short name path to you're script in the keyin editor. For more information see the MicroStation help. See "Ask Inga" for more detailed usage help. Wrapper for MicroStation command script generation.

classes

- *Layer*
- ***CommandFile***
  - *CommandFile.setup()*
  - *CommandFile.teardown()*
  - *CommandFile.cmd()*
  - *CommandFile.point()*
  - *CommandFile.points()*
  - *CommandFile.reset()*

- *CommandFile.line()*
- *CommandFile.polyline()*
- *CommandFile.circle()*
- *CommandFile.text()*
- *CommandFile.layer()*
- *CommandFile.set_layer()*
- *CommandFile.pop_layer()*
- *CommandFile.store()*
- *CommandFile.move()*
- *CommandFile.copy()*
- *CommandFile.rotate()*
- *CommandFile.scale()*
- *CommandFile.erase()*

Layer

class pygencad.microstation.**Layer**(*name*, *co=None*, *lw=None*, *lc=None*)

MicroStation level wrapper.

Sets the bylevel color, line-weight, and line-type to the values provided by the co, lw, and lc. The values default to "bylevel" which uses ustation's default/current settings.

Parameters

- **name** (*string OR Layer-like*) – The name visible in ustation after the script is run. A tuple of level properties, or a Layer object may be passed as the first argument (name) in which case the new level is initialized with provided values.
- **co** (*int*) – The color of the level.
- **lw** (*number*) – The line-weight of the level.
- **lc** (*number*) – The line-class (type) of the level.

CommandFile

class pygencad.microstation.**CommandFile**(*filelike*, *setup=None*, *teardown=None*)

Wrapper for MicroStation script generation.

The ustation wrapper should have additional methods for setting ACS and placing cells, mirroring the autocad module's methods.

Parameters

- **filelike** (*filelike*) – An object with a write method.
- **setup** (*string OR callable*) – Commands to include at the beginning of the script.
- **teardown** (*string OR callable*) – Commands to include at the end of the script.

class Layer(*name*, *co=None*, *lw=None*, *lc=None*)

A local binding to the layer class

CommandFile.**setup**()
Write default configuration info and run user provided setup func.

CommandFile.**teardown**()
Write cleanup commands and run user provided teardown func.

CommandFile.**point**(*p*, *reset=False*, *mode='xy'*, *write=True*)
Convert an iterable into a point of the correct format.

The mode argument defaults to 'xy=' absolute mode. Other modes can be specified, but currently no additional processing is performed.

Parameters

- **p** – The point to be converted.
- **reset** (*boolean*) – Optionally append the result of calling the reset method after writing the point.
- **mode** (*string*) – Optional point output mode.
- **write** (*boolean*) – Flag for writing output to script file.

Returns The point converted to a string.

CommandFile.**points**(*ps*, *reset=False*, *mode='xy'*)
Calls CommandFile.point on each member of an iterable.

Parameters

- **ps** – The points to be converted.
- **reset** (*boolean*) – Optionally append the result of calling the reset method after writing all points.
- **mode** (*string*) – Optional point output mode (unimplemented in base class).

CommandFile.**reset**(*write=True*)
Write a reset command.

Parameters **write** (*boolean*) – Flag for writing output to script file.

Returns The command that output by reset as a string.

CommandFile.**line**(*ps*, *reset=False*, *mode='xy'*)
Write the place line command.

Parameters

- **ps** – The end points of the lines.
- **reset** (*boolean*) – Optionally append the result of calling the reset method after writing all points.
- **mode** (*string*) – Optional point output mode.

CommandFile.**polyline**(*ps*, *close=True*, *reset=False*, *mode='xy'*)
Write the smartline command.

Support could be added in the future for drawing arc segments or other special smartline settings.

Parameters

- **ps** – The ordered points forming the line.
- **close** (*boolean*) – Flag indicating whether the line should end where it started.

- **reset** (*boolean*) – Optionally append the result of calling the reset method after writing all points.
- **mode** (*string*) – Optional point output mode.

CommandFile.**circle**(*center*, *radius*)
Write the place circle command.

The circle method uses "dx=" mode for setting the radius.

Parameters

- **center** (*iterable*) – Point in a format passable to CommandFile.point.
- **radius** (*number*) – Radius of the circle.

CommandFile.**text**(*text*, *p*, *height=None*, *ang=None*, *style=None*)
Write the place text command.

If a height is provided the text width is set to match. The "choose element" command is written after the text to avoid placing the same text accidentally.

Parameters

- **text** (*string*) – The text to place.
- **p** (*iterable*) – The insertion point in a format passable to CommandFile.point.
- **height** (*number*) – How tall the text lines are.
- **ang** (*number*) – The angle of the text in degrees.
- **style** (*string*) – The text style to activate before placing the text.

CommandFile.**store**(*name*, **els*)
Store provided elements to be manipulated later.

In MicroStation named groups are used. If 'l' is in els only the last element is added to the set. Otherwise the elements in the provided groups are joined into the named group.

Parameters

- **name** (*MicroStation quick select group*) – The id used to store the selection.
- ***els** (*previous group names, or 'l' for last element*) – The elements to store.

Raises ValueError if no elements are provided.

Returns An element group passable to move/copy/rotate/scale methods.

CommandFile.**move**(*els*, *base=(0, 0)*, *dest=(0, 0)*)
Transform the provided elements from base to dest.

Parameters

- **els** (*iterable, selection name, 'l' for last, or 'p' for previous*) – The elements to move.
- **base** (*iterable*) – The base point to transform from.
- **dest** (*iterable*) – The dest point to transform to.

Returns Reference to modified objects.

CommandFile.**copy**(*els*, *base=(0, 0)*, *dest=(0, 0)*)
Transform duplicates of the provided elements from base to dest.

Parameters

- **els** (*iterable, selection name, 'l' for last, or 'p' for previous*) – The elements to copy.
- **base** (*iterable*) – The base point to transform from.
- **dest** (*iterable*) – The dest point to transform to.

Returns Reference to duplicated objects.

CommandFile.**rotate**(*els*, *base=(0, 0)*, *ang=0*)

Rotate the provided elements from base by angle.

Parameters

- **els** (*iterable, selection name, 'l' for last, or 'p' for previous*) – The elements to rotate.
- **base** (*iterable*) – The base point to transform from.
- **ang** (*number*) – The angle to rotate in degrees (counter-clockwise).

Returns Reference to modified objects.

CommandFile.**scale**(*els*, *base=(0, 0)*, *scale=1*)

Scale the provided elements from base by scale.

Parameters

- **els** (*iterable, selection name, 'l' for last, or 'p' for previous*) – The elements to scale.
- **base** (*iterable*) – The base point to transform from.
- **scale** (*number*) – The scale factor to apply.

Returns Reference to modified objects.

CommandFile.**erase**(*els*)

Remove the indicated elements.

Parameters **els** (*iterable, selection name, 'l' for last, or 'p' for previous*) – The elements to remove.

CommandFile.**cmd**(*command*, **args*)

Write an arbitrary script command.

If the command string contains format expressions, and additional arguments were passed, the command string will be formated using the additional arguments.

```
>>> from pygencad.commandfile import CommandFile
>>> from StringIO import StringIO
>>> f = StringIO()
>>> with CommandFile(f) as script:
...     script.cmd("command with data: {:0.5f}", 355/113.0)
...
'1'
>>> print f.getvalue()
command with data: 3.14159
```

Parameters

- **command** (*string*) – The command to be written to the script.

- ***args** – Values to be formated by the command string.

Returns Some reference to any object created.

CommandFile.**layer**(**args*, ***kwds*)

Script layer context manager.

Activates the provided layer settings, and resets layer state on exit.

```
>>> from pygencad.commandfile import CommandFile
>>> from StringIO import StringIO
>>> f = StringIO()
>>> with CommandFile(f) as script:
...     with script.layer('layer'):
...         script.cmd("test")
...
'1'
>>> print f.getvalue()
Layer:
    lv=layer
    co=None
    lw=None
    lc=None
test
```

Parameters

- ***args** – Valid Layer arguments.
- ****kwargs** – Valid Layer arguments.

CommandFile.**pop_layer**()

Restores active layer prior to last set_layer call.

CommandFile.**set_layer**(*layer*, **args*, ***kwargs*)

Activates the provided layer object or description.

Parameters

- **layer** – A layer object, or a valid Layer name argument.
- ***args** – Valid Layer arguments.
- ****kwargs** – Valid Layer arguments.

SVG

The SVG module generates tags inside <svg> tags. It does not generate a doctype, or XML/HTML wrappers. The default extension is '.svg' but you will need to write appropriate info outside of the script block. The simplest wrapper to use is HTML5, where all that is required to embed the SVG tags in a minimal html doc, no doctype required!

```
>>> from pygencad import *
>>> from StringIO import StringIO
>>> f = StringIO()
>>> f.write("<html><head></head><body>\n")
>>> with get_script(f, 'svg') as script:
...     with script.layer('hello'):
...         hello = script.cmd('hello')
```

```
...         hello.text = "Hello, World!"
...
>>> f.write("\n</body></html>")
>>> print f.getvalue()
<html><head></head><body>
<svg ...><g ...><hello class="hello">Hello, World!</hello><style>
<![CDATA[
    line, polyline, circle {
        stroke: black;
        fill: none;
        stroke-width: 0.1px;
    }
    .hello {
            stroke: black;
            stroke-width: 0.1;
            stroke-dasharray: 0;
    }
]]>
</style></g></svg>
</body></html>
```

Backend that outputs to SVG tags.

SVG is generated using the 'xml.etree' ElementTree std. lib module. A write parameter has been added to all geometry creation methods, and all methods return any ELementTree.Elements created.

Note: If the svg.CommandFile isn't used as a context manager you will have to provide a root ElementTree.Element using CommandFile.set_root method. Also you will need to call CommandFile.teardown manually to add the style block to the root, and write the tree to the provided file object.

Warning: The point, points, and reset methods have no meaningful representation in SVG. Any script using these methods cannot be run against the SVG backend.

There are several special module attributes:

attr WIDTH SVG doc width.

attr HEIGHT SVG doc height.

attr FACTOR The initial scale factor of the drawing.

attr NAVIGATE Toggles inclusion of SVGPan.js in teardown.

WIDTH and HEIGHT are written in the <svg> tag as their respective attributes. The viewbox is set to '0 0 WIDTH HEIGHT' to match display and drawing units.

To better match other CAD packages a group is added inside the SVG tag wrapping all other elements that mirrors everything across the y-axis so positive y is up, and everything is translated down by HEIGHT so that the origin is in the bottom left corner. This is also where initial scale FACTOR is applied.

classes

- *Layer*
- ***CommandFile***

- *CommandFile.set_root()*
- *CommandFile.setup()*
- *CommandFile.teardown()*
- *CommandFile.set_layer()*
- *CommandFile.pop_layer()*
- *CommandFile.cmd()*
- *CommandFile.line()*
- *CommandFile.polyline()*
- *CommandFile.circle()*
- *CommandFile.text()*
- *CommandFile.layer()*
- *CommandFile.store()*
- *CommandFile.move()*
- *CommandFile.copy()*
- *CommandFile.rotate()*
- *CommandFile.scale()*
- *CommandFile.erase()*

Layer

class pygencad.svg.**Layer**(*name*, *co=None*, *lw=None*, *lc=None*)

Group objects with identical styles.

Note: All layers have a fill style of none.

Parameters

- **name** (*string OR Layer-like*) – This is the class name for the layer style. A tuple of layer properties, or a Layer object may be passed as the first argument (name) in which case the new layer is initialized with provided values.
- **co** – The color of the layer [default: black].
- **lw** – The line-weight of the layer [default: 0.1].
- **lc** – The line-class (type) of the layer. The line-class can be None for continuous lines, one of the names in the Layer.linecls class attribute, or an iterable of ints.

Store properties and setup template.

linecls = {'phantom': (0.75, 0.2, 0.2, 0.2, 0.2, 0.2), 'hidden': (0.25, 0.2), 'continuous': (0,), 'center': (1, 0.2, 0.2, 0.2)}

Named line classes

colors = {1: 'red', 2: 'yellow', 3: 'green', 4: 'cyan', 5: 'blue', 6: 'magenta', 7: 'black', 8: 'gray', 9: 'lightgray'}

Indexed colors

__str__()
Render this layer as a CSS style.

CommandFile

class pygencad.svg.**CommandFile**(*filelike*, *setup=None*, *teardown=None*)
Wrapper for SVG generation.

Convenience methods for building an SVG tree and writing it to a filelike.

Parameters

- **filelike** (*filelike*) – An object with a write method.
- **setup** (*callable OR valid CommandFile.cmd argument*) – Commands to include at the beginning of the script.
- **teardown** (*callable OR valid CommandFile.cmd argument*) – Commands to include at the end of the script.

class Layer(*name*, *co=None*, *lw=None*, *lc=None*)
A local binding to the SVG Layer class

CommandFile.**set_root**(*el*, *transform=False*)
Sets the ElementTree Element to which tags will be added.

Also adds a wrapper group that modifies the SVG coordinate system.

Parameters **el** (*ElementTree.Element*) – The element to set as root.

Raises TypeError if el doesn't pass ElementTree.iselement test.

CommandFile.**setup**()
Add default configuration info and run user provided setup func.

Called automatically by CommandFile context manager. Writes default setup code, then adds any user provided setup. If the user setup is callable the return value is written to the script, otherwise the user setup is written as a string.

Note: User setup is handled differently in SVG, if the user setup is not a function it is passed to the cmd method to be added as a tag.

CommandFile.**teardown**()
Add final tags, run user provided teardown func, and write filelike.

Called automatically by CommandFile context manager. Writes any user provided teardown, then writes default teardown code. If the user teardown is callable the return value is written to the script, otherwise the user teardown is written as a string.

Note: User teardown is handled differently in SVG, if the user teardown is not a function it is passed to the cmd method to be added as a tag.

CommandFile.**set_layer**(*layer*, **args*, ***kwargs*)
Activates the provided layer object or description.

Parameters

- **layer** – A layer object, or a valid Layer name argument.

- ***args** – Valid Layer arguments.
- ****kwargs** – Valid Layer arguments.

CommandFile.**pop_layer**()
Restores active layer prior to last set_layer call.

CommandFile.**cmd**(*command*, **args*, ***kwargs*)
Add arbitrary elements to script root.

The command argument is expected to be a tag name, or an ElementTree.Element, or an iterable of ElementTree Elements. If command is a string then *args and **kwargs are passed to the Element init. if command is already an element then the extra args are ignored.

Parameters

- **command** (*string OR ElementTree.Element OR Element iterable*) – The element(s) to be added to the document.
- ***args** (*dict*) – The attrib argument of ElementTree.Element.
- ****kwargs** – Extra attributes for the new ElementTree.Element.
- **write** (*boolean*) – Whether the element should be added to root (default=True).

Returns Element

CommandFile.**line**(*ps*, *reset=False*, *mode=None*, *write=True*)
Add a series of line segment tags to root.

Since the SVG line tag only supports a single line segment multiple tags may be added to a single set of points.

Parameters

- **ps** – The end points of the lines.
- **reset** (*boolean*) – Unused in SVG backend.
- **mode** (*string*) – Unimplemented, relative coordinates could be implemented for individual method invocations.
- **write** (*boolean*) – Whether the line elements should be added to root.

Returns Element list

CommandFile.**polyline**(*ps*, *close=True*, *reset=False*, *mode=None*, *write=True*)
Add a polyline tag to root.

Points are written as "x,y" pairs seperated by spaces, so splitting produces point strings, and splitting those strings on ',' produces the point components.

Parameters

- **ps** – The ordered points forming the line.
- **close** (*boolean*) – Flag indicating whether the line should end where it started.
- **reset** (*boolean*) – Unused in SVG backend.
- **mode** (*string*) – Unimplemented, relative coordinates could be implemented for individual method invocations.
- **write** (*boolean*) – Whether the element should be added to root.

Returns Element

CommandFile.**circle**(*center*, *radius*, *write=True*)
Add a circle tag to root.

Parameters

- **center** (*iterable*) – The center of the circle.
- **radius** (*number*) – The radius of the circle.
- **write** (*boolean*) – Whether the element should be added to root.

Returns Element

CommandFile.**text**(*text*, *p*, *height=1.0*, *ang=0*, *just='tl'*, *write=True*)
Add a text tag to root.

The text is inspected for newline chars, and <tspan> tags are added with (hopefully) appropriate dx/dy values to create multiline text.

Parameters

- **text** (*string*) – The text to place.
- **p** (*iterable*) – The insertion point in a format passable to CommandFile.point.
- **height** (*number*) – How tall the text lines are.
- **ang** (*number*) – The angle of the text in degrees.
- **just** (*string*) – The justification of the text, a pair of [t, m, b][l, c, r].
- **write** (*boolean*) – Whether the element should be added to root.

Returns Element

CommandFile.**store**(*name*, **els*)
Store provided elements to be manipulated later.

This method is provided as a uniform interface for grouping objects regardless of backend. In the SVG backend this is a noop.

Parameters

- **name** (*Unused*) – The id used to store the selection.
- ***els** (*SVG elements or element lists.*) – The elements to store.

Raises ValueError if no elements are provided.

Returns An element group passable to move/copy/rotate/scale methods.

CommandFile.**move**(*els*, *base=(0, 0)*, *dest=(0, 0)*)
Transform the provided elements from base to dest.

Parameters

- **els** (*iterable of Elements*) – The elements to move.
- **base** (*iterable*) – The base point to transform from.
- **dest** (*iterable*) – The dest point to transform to.

Returns Element list

CommandFile.**copy**(*els*, *base=(0, 0)*, *dest=(0, 0)*, *write=True*)
Transform duplicates of the provided elements from base to dest.

Parameters

- **els** (*iterable of Elements*) – The elements to copy.
- **base** (*iterable*) – The base point to transform from.
- **dest** (*iterable*) – The dest point to transform to.
- **write** (*boolean*) – Whether the elements should be added to root.

Returns Element list

CommandFile.**rotate**(*els*, *base=(0, 0)*, *ang=0*)
Rotate the provided elements from base by angle.

Parameters

- **els** (*iterable of Elements*) – The elements to rotate.
- **base** (*iterable*) – The base point to transform from.
- **ang** (*number*) – The angle to rotate in degrees (counter-clockwise).

Returns Element list

CommandFile.**scale**(*els*, *base=(0, 0)*, *scale=(1, 1)*)
Scale the provided elements from base by scale.

Remember that scaling an axis by -1 is equivalent to mirroring.

Parameters

- **els** (*iterable of Elements*) – The elements to scale.
- **base** (*iterable*) – The base point to transform from.
- **scale** (*iterable*) – The x and y scale factor to apply.

Returns Element list

CommandFile.**erase**(*els*)
Remove the indicated elements from root.

Parameters **els** (*iterable of Elements*) – The elements to remove.

Returns Element list

CommandFile.**layer**(**args*, ***kwds*)
Script layer context manager.

Activates the provided layer settings, and resets layer state on exit.

```
>>> from pygencad.commandfile import CommandFile
>>> from StringIO import StringIO
>>> f = StringIO()
>>> with CommandFile(f) as script:
...     with script.layer('layer'):
...         script.cmd("test")
...
'l'
>>> print f.getvalue()
Layer:
    lv=layer
    co=None
    lw=None
    lc=None
test
```

Parameters

- **`*args`** – Valid Layer arguments.
- **`**kwargs`** – Valid Layer arguments.

Python Module Index

p

Index

Symbols

B

C

E

G

L

M

P

R

S

T

U