
PyX Manual

Release 0.18

Jörg Lehmann, Michael Schindler, André Wobst

2026/10/18

CONTENTS

1	Introduction	3
1.1	What is PyX?	3
1.2	A first example	3
1.3	Organization of this manual	4
1.4	Installation	4
1.5	Configuration and file location	5
2	Basic graphics	9
2.1	Introduction	9
2.2	Path operations	11
2.3	Attributes: Styles and Decorations	12
2.4	The attribute system: <i>attr</i>	14
2.5	Styles: <i>style</i>	16
2.6	Decorators: <i>deco</i>	18
3	Paths	23
3.1	Constructing paths	23
3.2	Path items	24
3.3	Class <i>path</i>	25
3.4	Normalized paths: <i>normpath</i>	27
3.5	Predefined shapes	30
4	MetaPost-like paths	31
4.1	Example	31
4.2	Path construction: <i>path</i>	32
4.3	Knots	32
4.4	Links	33
5	Deformers	35
5.1	Corner smoothing: <i>smoothed</i>	35
5.2	Parallel offset curves: <i>parallel</i>	36
5.3	Spring / coil paths: <i>cycloid</i>	37
6	Canvas	39
6.1	Working with canvases	39
6.2	Class <i>canvas</i>	40
6.3	Class <i>clip</i>	42
7	Documents	43
7.1	Creating single- and multi-page documents	43
7.2	Class <i>page</i>	43

7.3	Class <i>paperformat</i>	44
7.4	Class <i>document</i> and output writers	44
8	Text	47
8.1	Overview and Quick Start	47
8.2	Architecture of the TeX Interface	47
8.3	TeX and LaTeX Engines	48
8.4	Module-Level Functions	53
8.5	TeX Output Parsers and Error Handling	53
8.6	Text Attributes (Alignment, Parboxes, Math Mode, Font Size)	56
8.7	UnicodeEngine: Direct Font Rendering without TeX	59
8.8	Using the LaTeX <i>graphics</i> Bundle with PyX	59
8.9	Configuration and Debugging	60
9	Graphs	63
9.1	Introduction	63
9.2	Component Architecture and Canvas Layers	65
9.3	Module <i>graph.graph</i> : Graph Geometry	66
9.4	Module <i>graph.data</i> : Graph Data Sources	70
9.5	Module <i>graph.style</i> : Graph Styles	72
9.6	Module <i>graph.key</i> : Graph Keys (Legends)	76
10	Axes	77
10.1	Component Architecture	77
10.2	Module <i>graph.axis.axis</i> : Axis Classes	78
10.3	Module <i>graph.axis.timeaxis</i> : Date and Time Axes	81
10.4	Module <i>graph.axis.tick</i> : Axis Ticks	81
10.5	Module <i>graph.axis.parter</i> : Axis Partitioners	82
10.6	Module <i>graph.axis.texter</i> : Axis Texters	83
10.7	Module <i>graph.axis.painter</i> : Axis Painters	84
10.8	Module <i>graph.axis.rater</i> : Axis Raters	86
10.9	Module <i>graph.axis.positioner</i> : Axis Positioners	86
11	Module <i>box</i>: Convex box handling	87
11.1	Class <i>polygon</i>	87
11.2	Aligning and Tiling Lists of Boxes	88
11.3	Class <i>rect</i>	89
12	Module <i>connector</i>	91
12.1	Geometry, Angles, and Bulges	91
12.2	Connector Classes	92
13	Module <i>epsfile</i>: EPS file inclusion	95
13.1	Class <i>epsfile</i>	95
14	Module <i>pdffile</i>: PDF file inclusion	97
14.1	Class <i>pdffile</i>	97
15	Module <i>svgfile</i>: SVG file inclusion	99
15.1	Class <i>svgfile</i>	99
15.2	Unparsed vs. Parsed Mode	100
15.3	Coordinate Resolution (<i>resolution</i>)	100
16	Bitmaps	101
16.1	Introduction	101

16.2	Image Containers and File Readers	102
16.3	Placing Bitmaps on a Canvas	102
17	Module <i>bbox</i>	105
17.1	Constructing Bounding Boxes	105
17.2	Set Operations and Methods	105
18	Module <i>color</i>	107
18.1	Color Models and Named Colors	107
18.2	Color Gradients	108
18.3	Transparency	109
19	Module <i>pattern</i>	111
19.1	Class <i>pattern</i>	111
19.2	Predefined Hatching Patterns	112
20	Module <i>unit</i>	113
20.1	Configuring Units and Scaling Factors	113
20.2	Class <i>length</i>	114
20.3	Predefined Length Constants	114
20.4	Conversion Functions	115
21	Module <i>trafo</i>: Linear transformations	117
21.1	Mathematical Definition and Composition	117
21.2	Class <i>trafo</i>	117
21.3	Standard Transformation Subclasses	118
22	Appendix: Named colors	121
23	Appendix: Named gradients	123
24	Appendix: Path styles in <i>style</i> module	125
25	Appendix: Arrows in <i>deco</i> module	127
26	Module <i>utils</i>: Helper utilities	129
	Python Module Index	131
	Index	133

Author

Jörg Lehmann <joerg@pyx-project.org>, Michael Schindler <m-schindler@users.sourceforge.net>, André Wobst <wobsta@pyx-project.org>

Date

2026/10/18

Abstract

PyX is a Python package for the creation of PostScript, PDF, and SVG files. It combines an abstraction of the PostScript drawing model with a TeX/LaTeX interface and direct Type 1 / OpenType font support. Complex tasks such as 2D and 3D publication-quality data plots in cartesian or curvilinear coordinates, path deformations, and box-and-connector diagrams are built on top of these primitives in a modular, extensible architecture.

INTRODUCTION

1.1 What is PyX?

PyX is a Python package for creating publication-quality graphics in PostScript, PDF, and SVG formats. It combines a clean Pythonic abstraction of the PostScript drawing model with seamless TeX/LaTeX integration and native Type 1 and OpenType font handling.

The design of PyX is guided by several core principles:

- **Programmatic precision:** Figures are described in Python code rather than assembled interactively in a GUI. This makes complex or data-driven figures reproducible, parameterizable, and easy to maintain alongside scientific or engineering code.
- **Unified drawing primitives:** Geometric shapes are represented as math-aware PostScript-like paths (*path* and *normpath*) or MetaPost-like spline paths (*metapost.path*) that support intersections, arc-length parameterization, tangents, curvatures, and geometric deformations (*deformer*).
- **Composable canvases and attributes:** Graphics primitives are collected on hierarchically nestable canvases (*canvas*) and styled through a typed, composable attribute system (*attr*, *style*, *deco*, *color*, *pattern*, *trafo*).
- **First-class typography:** Text can be typeset through an interactive TeX or LaTeX subprocess (*text.TextEngine*, *text.LatexEngine*) so that mathematical formulas and document fonts match your manuscript, or rendered directly via *text.UnicodeEngine* using Type 1 or OpenType fonts without invoking TeX.
- **Modular plotting architecture:** High-level components such as 2D and 3D graphs (*graph*) and axes (*graph.axis*) are built directly on top of PyX's path, canvas, and text primitives, with clear interfaces for custom data sources, plot styles, axis partitioners, texters, and painters.

1.2 A first example

To see how these concepts fit together, consider a minimal script that draws a stroked red rectangle and typesets a centered mathematical expression inside it:

```
from pyx import *

c = canvas.canvas()
c.stroke(path.rect(0, 0, 4, 2), [style.linewidth.Thick, color.rgb.red])
c.text(2, 1, r"$\int_{-\infty}^{\infty} e^{-x^2} \backslash, \mathrm{d}x = \sqrt{\pi}$",
      [text.halign.center, text.valign.middle])

c.writeEPSfile("hello")
c.writePDFfile("hello")
c.writeSVGfile("hello")
```

Here `canvas.canvas()` creates an drawing surface, `path.rect(0, 0, 4, 2)` constructs a rectangular path of width 4 cm and height 2 cm (using PyX's default user unit of centimeters), and the attribute list `[style.linewidth.Thick, color.rgb.red]` specifies how the path is stroked. Finally, `writeEPSfile`, `writePDFfile`, and `writeSVGfile` write the canvas to `hello.eps`, `hello.pdf`, and `hello.svg`.

1.3 Organization of this manual

This manual is organized both as a progressive guide for new users and as a complete technical reference for experienced PyX users and developers:

- **Basic graphics** (*Basic graphics, Paths, MetaPost-like paths, Deformers, Canvas, Documents*): Introduces the drawing model, the attribute system (`attr`), stroke and fill styles (`style`), path decorators (`deco`), path construction and normalized path geometry (`path`, `normpath`, `metapost`), path deformers (`deformer`), canvases (`canvas`), and multi-page documents (`document`).
- **Text** (*Text*): Covers text typesetting with TeX, LaTeX, and `text.UnicodeEngine`, text alignment and sizing attributes, color and transformation handling, markers, and diagnostics.
- **Plotting** (*Graphs, Axes*): Describes 2D and 3D data plotting, data sources (`graph.data`), plotting styles (`graph.style`), graph keys (`graph.key`), and the axis system (`graph.axis`, `graph.axis.parter`, `graph.axis.texter`, `graph.axis.painter`, `graph.axis.rater`, `graph.axis.tick`).
- **Boxes and Connectors** (*Module box: Convex box handling, Module connector*): Explains aligned polygonal and rectangular boxes and the connectors linking them.
- **External graphics and bitmaps** (*Module epsfile: EPS file inclusion, Module pdffile: PDF file inclusion, Module svgfile: SVG file inclusion, Bitmaps*): Covers embedding EPS, PDF, SVG, and raster bitmap images.
- **Basic classes and modules** (*Module bbox, Module color, Module pattern, Module unit, Module trafo: Linear transformations*): Documents bounding boxes, color spaces and gradients, fill patterns, the multi-scale unit system, and affine transformations.
- **Appendix**: Provides visual catalogs of predefined colors, gradients, path styles, arrows, and internal utility helpers.

Note

Throughout this manual, classes and functions that accept geometric coordinates or lengths in PyX units (which support unit scaling and explicit units such as `2*unit.t_cm`) often have a companion class or method ending in `_pt` (such as `path.line_pt` or `canvas.canvas.text_pt()`). These `_pt` variants take numerical values directly in PostScript points (1 pt = 1/72 inch) and bypass unit conversion for performance-critical internal and developer code.

1.4 Installation

1.4.1 Prerequisites

PyX requires Python 3. When using the default TeX/LaTeX text engines (`text.TextEngine` and `text.LatexEngine`), a working TeX installation (such as TeX Live or MiKTeX) with Type 1 fonts is required. If you use `text.UnicodeEngine` exclusively with shipped or local Type 1 / OpenType fonts, PyX can generate PostScript, PDF, and SVG output without invoking an external TeX interpreter.

To include or manipulate raster images beyond uncompressed PNM or JPEG files, the Python Imaging Library (`Pillow`) is optional but recommended.

1.4.2 Installing PyX

The easiest way to install the latest release of PyX is via `pip`:

```
pip install pyx
```

To install PyX from a source distribution or a Git checkout, run:

```
pip install .
```

from the top-level directory of the source tree.

By default, PyX installs as a pure Python package and locates TeX files and fonts by invoking the `kpsewhich` command-line tool. For faster file lookup in environments with frequent font searches, PyX also includes an optional C extension module, `pykpathsea`, which links directly against `libkpathsea`. You can enable building `pykpathsea` by editing `setup.cfg` prior to installation and specifying the appropriate include and library paths for your TeX distribution.

1.5 Configuration and file location

1.5.1 Runtime configuration (pyxrc)

PyX reads its runtime configuration from a sequence of INI-style configuration files:

1. The built-in default configuration `pyx/data/pyxrc` shipped with PyX.
2. A user-specific configuration file: `~/.pyxrc` on Unix/macOS systems, or `%APPDATA%\pyxrc` on Windows.
3. Any configuration file specified by the environment variable `PYXRC`.

Settings in later files override earlier ones. The configuration is divided into three sections:

[general]

- `space` (default `SPACE`): Whitespace-separated list options in `pyxrc` use this token to represent literal space characters inside list items.

[text]

- `defaulttextengine` (default `TexEngine`): Default text engine class initialized in `text` (`TexEngine`, `LatexEngine`, or `UnicodeEngine`).
- `tex` (default `tex`): Command (and optional whitespace-separated arguments) used to start the plain TeX interpreter.
- `latex` (default `latex`): Command (and optional whitespace-separated arguments) used to start the LaTeX interpreter.
- `psfontmaps` (default `psfonts.map`): Whitespace-separated list of font mapping files used for PostScript and SVG output.
- `pdffontmaps` (default `pdftex.map`): Whitespace-separated list of font mapping files used for PDF output.
- `wait` (default `60`): Maximum time in seconds to wait for a response from the TeX/LaTeX subprocess before raising a timeout error.
- `showwait` (default `5`): Interval in seconds between progress warnings logged while waiting for TeX/LaTeX (set to `0` to disable).
- `texipc` (default `0`): Boolean flag enabling TeX's `--ipc` option for reading DVI pages before TeX terminates.
- `chroot` and `tmpdir`: Optional path overrides for running TeX inside a chroot environment or a custom temporary directory.

[filelocator]

- **methods** (default `local internal pykpathsea kpsewhich`): Whitespace-separated list of file lookup methods tried in order when PyX searches for fonts, metrics, headers, or figures:
 - `local`: Searches the current working directory.
 - `internal`: Searches the `pyx/data` package resources shipped with PyX (such as built-in `.lfs` files, `pyxrc`, and standard AFM/Type 1 files).
 - `recursivedir`: Recursively searches the directories listed in the `recursivedir` option of `[filelocator]`.
 - `ls-R`: Searches TeX `ls-R` filename databases listed in the `ls-R` option of `[filelocator]`.
 - `pykpathsea`: Uses the compiled `pykpathsea` C extension if available.
 - `kpsewhich`: Calls the `kpsewhich` executable (customizable via the `kpsewhich` option in `[filelocator]`).
 - `locate`: Calls the system `locate` command (customizable via the `locate` option in `[filelocator]`).

1.5.2 Developer interface: *config* and logging

For PyX developers, the *config* module provides programmatic access to configuration settings, resource discovery, and subprocess execution:

`config.get(section, option, default=...)`

`config.getint(section, option, default=...)`

`config.getfloat(section, option, default=...)`

`config.getboolean(section, option, default=...)`

`config.getlist(section, option, default=...)`

Retrieve an option from *section* as a string, integer, float, boolean, or whitespace-split list of strings. If *default* is provided, it is returned whenever the section or option is not present in the configuration.

`config.open(filename, formats, ascii=False)`

Locate and open *filename* by trying the configured `[filelocator]` methods for the given list of *format* specifications. Returns a binary stream by default, or a text stream with `ascii` encoding and `surrogateescape` error handling when *ascii* is `True`. Raises `IOError` if the file cannot be found.

Standard file formats defined on *format* are:

- `format.tfm (.tfm)`
- `format.afm (.afm)`
- `format.pfm (.pfm)`
- `format.fontmap (.map)`
- `format.pict (.eps, .epsi)`
- `format.tex_ps_header` (PostScript headers and `.enc` encoding files)
- `format.type1 (.pfa, .pfb)`
- `format.vf (.vf)`
- `format.dvips_config`
- `format.ttf (.ttf, .ttc, .TTF, .TTC, .dfont)`
- `format.t42 (.t42, .T42)`

- `format.otf(.otf)`
- `format.pyx` (internal PyX data files)

PyX uses Python's standard logging module to report warnings and diagnostic information:

- `logging.getLogger("pyx")`: Main logger used for warnings across all PyX modules.
- `logging.getLogger("pyx.execute")`: Logs external subprocess invocations (such as `tex`, `latex`, `kpsewhich`, or `gs`) at the INFO level.
- `logging.getLogger("pyx.filelocator")`: Logs every file lookup and its resolved path at the INFO level.

To inspect which external commands and font files PyX uses during a run, enable INFO logging in your script:

```
import logging
logging.basicConfig(level=logging.INFO)
```


BASIC GRAPHICS

The chapters in this part describe the core building blocks for creating graphics in PyX and writing the result to PostScript, EPS, PDF, and SVG files. Before diving into the reference chapters for individual modules, this chapter introduces PyX's drawing model, path construction and geometric operations, the attribute system (*attr*), path styles (*style*), and path decorators (*deco*).

2.1 Introduction

The *path* module allows you to construct PostScript-like *paths*, which are the primary geometric building blocks for drawings in PyX. A path represents an arbitrary shape composed of straight line segments, circular or elliptical arc segments, and cubic Bézier curves. A path does not need to be contiguous; it may comprise several disconnected segments called *subpaths*.

Typically, a path is constructed by passing path primitives (*path.moveto*, *path.lineto*, *path.curveto*, etc.) to the *path.path* constructor. For example, the following snippet defines a path *p* consisting of a straight line from (0, 0) to (1, 1):

```
from pyx import *
p = path.path(path.moveto(0, 0), path.lineto(1, 1))
```

Equivalently, you can use the predefined *path.path* subclass *path.line*:

```
p = path.line(0, 0, 1, 1)
```

While geometric operations can already be performed on a path object itself (see the next section), rendering the path requires a *canvas.canvas* instance, conventionally named *c*:

```
c = canvas.canvas()
```

To draw the path on the canvas and write the figure to an EPS, PDF, or SVG file, use *canvas.canvas.stroke()* followed by one of the output methods:

```
c.stroke(p)
c.writeEPSfile("line")
c.writePDFfile("line")
c.writeSVGfile("line")
```

The file extension (*.eps*, *.pdf*, or *.svg*) is appended automatically if not already present in the filename.

As a second example, let us define a path consisting of two disconnected subpaths:

```
cross = path.path(path.moveto(0, 0), path.rlineto(1, 1),
                  path.moveto(1, 0), path.rlineto(-1, 1))
```

The first subpath is a straight line from (0, 0) to (1, 1) using `path.rlineto`, whose coordinates are measured relative to the current point. The second `path.moveto` opens a new subpath starting at (1, 0) and ending at (0, 1). Even though the two lines intersect at (1/2, 1/2), they remain separate subpaths. In general, every occurrence of `path.moveto` starts a new subpath. Consequently, a rectangle should not be constructed as four separate subpaths:

```
rect1 = path.path(path.moveto(0, 0), path.lineto(0, 1),
                  path.moveto(0, 1), path.lineto(1, 1),
                  path.moveto(1, 1), path.lineto(1, 0),
                  path.moveto(1, 0), path.lineto(0, 0))
```

which produces unjoined corners (see Fig. *Rectangle consisting of (a) four separate lines, (b) one open path, and (c) one closed path. (d) Filling a path closes open subpaths automatically.* a). Keeping the pen down between corners produces a single open subpath (Fig. *Rectangle consisting of (a) four separate lines, (b) one open path, and (c) one closed path. (d) Filling a path closes open subpaths automatically.* b):

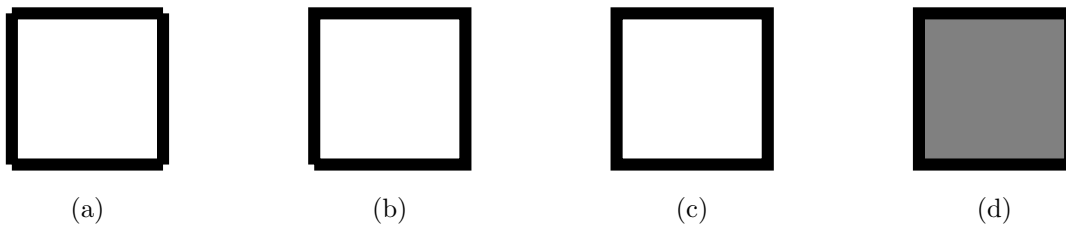


Fig. 1: Rectangle consisting of (a) four separate lines, (b) one open path, and (c) one closed path. (d) Filling a path closes open subpaths automatically.

```
rect2 = path.path(path.moveto(0, 0), path.lineto(0, 1),
                  path.lineto(1, 1), path.lineto(1, 0),
                  path.lineto(0, 0))
```

As visible in the lower-left corner of Fig. *Rectangle consisting of (a) four separate lines, (b) one open path, and (c) one closed path. (d) Filling a path closes open subpaths automatically.* b), an open path whose start and end points coincide still has two unconnected line caps at its origin. To properly join the start and end of the subpath, close it with `path.closepath` (Fig. *Rectangle consisting of (a) four separate lines, (b) one open path, and (c) one closed path. (d) Filling a path closes open subpaths automatically.* c):

```
rect3 = path.path(path.moveto(0, 0), path.lineto(0, 1),
                  path.lineto(1, 1), path.lineto(1, 0),
                  path.closepath())
```

`path.closepath` adds a straight line segment from the current point back to the initial point of the current subpath and joins the meeting edges using the active line join style. Filling a path implicitly closes every open subpath, as shown in Fig. *Rectangle consisting of (a) four separate lines, (b) one open path, and (c) one closed path. (d) Filling a path closes open subpaths automatically.* d):

```
c.stroke(rect2, [deco.filled([color.grey(0.5)])])
```

Here, the second argument to `canvas.canvas.stroke()` is an attribute list containing the decorator `deco.filled`, which instructs PyX to fill the interior of the stroked path with the given gray shade. Rectangles are, of course, also available directly as `path.rect`:

```
rect = path.rect(0, 0, 1, 1)
```

where the first two arguments specify the lower-left corner and the last two arguments specify the width and height. Full details on all path items and predefined shapes are given in chapter *Paths* (and smooth Hobby–Knuth splines in

chapter *MetaPost-like paths*).

2.2 Path operations

Often, you want to perform geometric calculations on paths before stroking or filling them—such as intersecting two paths, splitting them at their intersection points, and joining the resulting segments to form a new closed contour.

Suppose you want to draw the radii from the center of a circle to its intersection points with a straight line. The following code produces Fig. *Intersection of a circle with a straight line yielding two radii*:

```
from pyx import *

c = canvas.canvas()

circle = path.circle(0, 0, 2)
line = path.line(-3, 1, 3, 2)
c.stroke(circle, [style.linewidth.Thick])
c.stroke(line, [style.linewidth.Thick])

isects_circle, isects_line = circle.intersect(line)
for isect in isects_circle:
    isectx, isecty = circle.at(isect)
    c.stroke(path.line(0, 0, isectx, isecty))

c.writePDFfile()
```

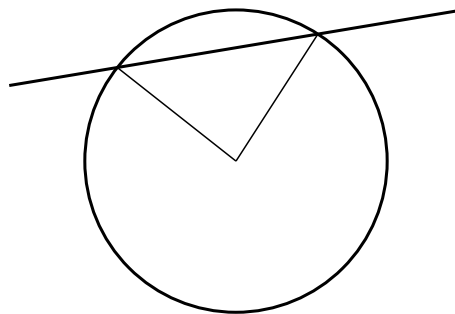


Fig. 2: Intersection of a circle with a straight line yielding two radii.

After defining a circle centered at $(0, 0)$ with radius 2 and a straight line, calling `circle.intersect(line)` returns a 2-tuple of lists of parameter values: the first list (*isects_circle*) contains the intersection parameters along *circle*, and the second (*isects_line*) contains the corresponding parameters along *line*. Passing each parameter to `path.path.at()` yields the (x, y) coordinates of the intersection point.

To shade the circular segment bounded by the line and the shorter arc (see Fig. *Intersection of a circle with a line yielding two radii and a filled circular segment*), we split both paths at their intersection parameters and join the pieces:

```
arc1, arc2 = circle.split(isects_circle)
if arc1.arclen() < arc2.arclen():
    arc = arc1
else:
    arc = arc2
```

(continues on next page)

(continued from previous page)

```

isects_line.sort()
line1, line2, line3 = line.split(isects_line)

segment = line2 << arc

```

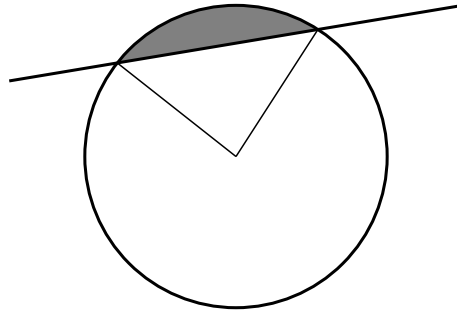


Fig. 3: Intersection of a circle with a line yielding two radii and a filled circular segment.

Splitting the closed circle at two points yields two arc segments, and `path.path.arclen()` selects the shorter one. Because `path.path.split()` requires an ascending list of parameter values, `isects_line` is sorted before splitting the line into three pieces. Finally, the `<<` operator joins the middle line segment `line2` and `arc` into a single connected subpath so that filling works as expected (whereas `+` would merely concatenate them as separate subpaths).

When working with path parameters, note that PyX converts paths into normalized paths (`normpath.normpath`) composed of straight lines and cubic Bézier curves before performing geometric operations. Each elementary segment of a normalized subpath spans a parameter interval of length 1 (see `normpath.normpathparam`), and the start and end parameters of a path are returned by `path.path.begin()` and `path.path.end()`. Alternatively, all path methods that accept a parameter value also accept a PyX length specifying the arc length measured from the start of the path:

```

from math import pi
r = 2
pt1 = path.circle(0, 0, r).at(r * pi)
pt2 = path.circle(0, 0, r).at(r * 3 * pi / 2)
c.stroke(path.line(pt1[0], pt1[1], pt2[0], pt2[1]))

```

2.3 Attributes: Styles and Decorations

Attributes control the visual appearance and geometric processing of objects when they are drawn. Orthogonal properties—such as line width, line style, color, and arrowheads—are represented by distinct attribute classes and passed together in a list as an optional argument to drawing methods:

```
c.stroke(p, [style.linewidth.Thick, style.linestyle.dashed])
```

In fact, `canvas.canvas.stroke()` and `canvas.canvas.fill()` are convenient shortcuts for `canvas.canvas.draw()`: `c.stroke(p, attrs)` is equivalent to `c.draw(p, [deco.stroked] + list(attrs))`, and `c.fill(p, attrs)` is equivalent to `c.draw(p, [deco.filled] + list(attrs))`.

PyX's attribute system provides three key mechanisms:

1. **Predefined attribute instances:** Many attribute classes attach commonly used instances as class attributes. For example, `style.linewidth.Thick` is a predefined `style.linewidth` instance with width 0.04 w cm (twice `style.linewidth.normal`).

2. **Attribute merging and clearing:** When an attribute list is merged (for example, when user attributes are appended to default attributes), later instances of an exclusive attribute family automatically override earlier ones:

```
# The following two calls are equivalent:
c.stroke(p, [style.linewidth.Thick, style.linewidth.thin])
c.stroke(p, [style.linewidth.thin])
```

Every exclusive attribute class also provides a `.clear` attribute that removes any preceding instance of that class from the attribute list:

```
# The following two calls are equivalent:
c.stroke(p, [style.linewidth.Thick, style.linewidth.clear])
c.stroke(p)
```

Furthermore, `attr.clear` removes *all* preceding attributes from an attribute list.

3. **Customizing predefined instances by calling them:** Predefined decorator and deformer instances can be called with new arguments to return a modified copy. For example, `deco.filled` is already an instance of the fill decorator, and calling `deco.filled([color.rgb.green])` creates a new fill decorator with a local green fill color:

```
c.stroke(p, [deco.filled([color.rgb.green])])
```

Similarly, calling `deco.earrow(angle=30)` or `deco.earrow.Large([color.rgb.red])` returns a customized end-arrow decorator.

The main attribute categories in PyX are summarized below:

Attribute category	Description	Examples
<code>deco.deco</code>	Path decorators specifying how a path is drawn or ornamented	<code>deco.stroked</code> , <code>deco.filled</code> , <code>deco.arrow</code> , <code>deco.text</code>
<code>style.strokestyle</code>	Styles used when stroking a path	<code>style.linecap</code> , <code>style.linejoin</code> , <code>style.miterlimit</code> , <code>style.dash</code> , <code>style.linestyle</code> , <code>style.linewidth</code> , <code>color.color</code> , <code>color.transparency</code>
<code>style.fillstyle</code>	Styles used when filling a path	<code>color.color</code> , <code>color.transparency</code> , <code>pattern.pattern</code> , <code>style.fillrule</code>
baseclasses.deformer Geometric path deformers		<code>deformer.cycloid</code> , <code>deformer.smoothed</code> , <code>deformer.parallel</code>
<code>text.textattr</code>	Attributes used for text typesetting	<code>text.halign</code> , <code>text.valign</code> , <code>text.vshift</code> , <code>text.mathmode</code> , <code>text.phantom</code> , <code>text.size</code> , <code>text.parbox</code>
<code>trafo.trafo_pt</code>	Affine coordinate transformations	<code>trafo.mirror</code> , <code>trafo.rotate</code> , <code>trafo.scale</code> , <code>trafo.slant</code> , <code>trafo.translate</code>

2.4 The attribute system: *attr*

Whenever a list of attributes is passed to a PyX method, the *attr* module checks that only allowed attribute classes are present, resolves overrides and clearings, and enforces ordering constraints. It also provides *changeable attributes* that cycle through attribute lists when plotting multiple datasets in *graph*.

class attr.attr

Base class of all attributes in PyX (colors, decorators, styles, deformers, text attributes, and transformations). Its

`merge()` method appends `self` to an attribute list `attrs` and returns the updated list.

class `attr.exclusiveattr(exclusiveclass)`

Base class for attributes that swallow all preceding attributes in an attribute list that are instances of `exclusiveclass`.

class `attr.sortbeforeattr(beforetheclases)`

Base class for attributes that place themselves immediately before the first attribute in the list that is an instance of any class in `beforetheclases`.

class `attr.sortbeforeexclusiveattr(exclusiveclass, beforetheclases)`

Combines `exclusiveattr` and `sortbeforeattr`: removes all other instances of `exclusiveclass` and places itself before any instance of `beforetheclases`.

class `attr.clearclass(clearclass)`

Creates an attribute (customarily bound as `SomeAttr.clear`) whose `merge()` method removes all preceding instances of `clearclass` from an attribute list.

attr.clear

An attribute instance that removes *all* preceding attributes in an attribute list when merged.

class `attr.changeattr`

Base class for changeable attributes resolved via `select(index, total)()` when plotting multiple datasets in a graph.

class `attr.changelist(attrs, cyclic=1)`

A `changeattr` initialized with a list `attrs` of attribute choices:

- `attrs`: List of attributes to step through as datasets are plotted.
- `cyclic`: If true (default 1), selection wraps around modulo `len(attrs)` after reaching the end of the list; otherwise, selecting an index beyond `len(attrs) - 1` returns `None`.

For example, to cycle through three colors across curves in a graph:

```
mycolors = attr.changelist([color.rgb.red, color.rgb.green, color.rgb.blue])
```

class `attr.multichangeattr(changeattrs)`

A `changeattr` initialized with a dictionary or list `changeattrs` of `changeattr` instances indexed by the total number `total` of selections, delegating `select(index, total)` to `changeattrs[total].select(index, total)`.

attr.mergeattrs(attrs)

Merges the attribute list `attrs` by calling each attribute's `merge()` method in sequence and returns the resulting list.

attr.getattrs(attrs, getclasses)

Returns a list of all attributes in `attrs` that are instances of any class in `getclasses`.

attr.checkattrs(attrs, allowedclasses)

Checks that every element in `attrs` is an instance of at least one class in `allowedclasses`, raising `TypeError` otherwise.

attr.refineattrs(attrs, defaults, allowedclasses, allow_none=True)

Combines validation and merging: returns `None` if `allow_none` is true and `attrs` is `None`, otherwise validates `attrs` against `allowedclasses` and returns `mergeattrs(defaults + attrs)`.

attr.selectattrs(attrs, index, total)

attr.selectattr(attr, index, total)

Evaluates all `changeattr` instances in `attrs` (or a single `attr`) for 0-based selection `index` out of `total` items.

2.5 Styles: *style*

The *style* module provides classes that set PostScript, PDF, and SVG stroke and fill state parameters. Every style class provides a `.clear` attribute (an *attr.clearclass* instance) to remove preceding instances of that style from an attribute list.

2.5.1 Stroke styles

All stroke styles inherit from `strokestyle`:

class `style.linecap(value=0)`

Controls the shape of the ends of open subpaths when they are stroked:

- `linecap.butt` (0, default): The stroke is squared off flush at the endpoint of the path with no projection beyond it.
- `linecap.round` (1): A semicircular arc with a diameter equal to the line width is drawn around the endpoint.
- `linecap.square` (2): The stroke continues beyond the endpoint for half the line width and is squared off.
- `linecap.clear`: Clears any preceding *linecap* setting.

class `style.linejoin(value=0)`

Controls the shape of corners where two path segments meet when stroked:

- `linejoin.miter` (0, default): The outer edges of the two segments are extended until they meet at a sharp point (subject to *miterlimit*).
- `linejoin.round` (1): A circular arc with a diameter equal to the line width rounds off the corner.
- `linejoin.bevel` (2): The meeting segments are finished with butt caps and the resulting notch is filled with a triangle.
- `linejoin.clear`: Clears any preceding *linejoin* setting.

class `style.miterlimit(value=10.0)`

Limits the ratio of the miter length to the line width when `linejoin.miter` is active. When two segments meet at a sharp interior angle φ such that $1/\sin(\varphi/2)$ exceeds *value*, the corner is beveled instead of mitered. Predefined instances correspond to minimum interior angles before beveling occurs:

- `miterlimit.lessthan180deg` ($1 / \sin(\pi * 180 / 360) = 1.0$)
- `miterlimit.lessthan90deg` ($1 / \sin(\pi * 90 / 360)$)
- `miterlimit.lessthan60deg` ($1 / \sin(\pi * 60 / 360) = 2.0$)
- `miterlimit.lessthan45deg` ($1 / \sin(\pi * 45 / 360)$)
- `miterlimit.lessthan11deg` (10, default, corresponding to approximately 11.4783°)
- `miterlimit.clear`: Clears any preceding *miterlimit* setting.

class `style.dash(pattern=[], offset=0, relengths=1)`

Defines a dash pattern for stroked paths:

- *pattern*: List or tuple of numbers specifying alternating lengths of dashes and gaps.
- *offset*: Phase offset at which the dash pattern starts (default 0).
- *relengths*: When true (default 1), dash and gap lengths in *pattern* are scaled relative to the current line width (in units of the default line width); when false (0), they are interpreted in raw PostScript points.
- `dash.clear`: Clears any preceding *dash* setting.

class `style.linestyle(c=linecap.butt, d=dash([]))`

Combines a *linecap* `c` and a *dash* pattern `d`:

- `linestyle.solid`: `linecap.butt` with `dash([])`.
- `linestyle.dashed`: `linecap.butt` with `dash([2])`.
- `linestyle.dotted`: `linecap.round` with `dash([0, 2])`.
- `linestyle.dashdotted`: `linecap.round` with `dash([0, 2, 2, 2])`.
- `linestyle.clear`: Clears any preceding *linestyle* setting.

class `style.linewidth(width)`

Sets the stroke width to the PyX length *width* (using the w-scale if a plain number is given). Predefined line widths form a geometric progression around `linewidth.normal` ($0.02 * \text{unit.w_cm}$, approximately 0.567 pt) with factor $\sqrt{2}$:

- `linewidth.THIN`: $\text{normal} / \sqrt{32} = \text{normal} / 4\sqrt{2}$
- `linewidth.THIn`: $\text{normal} / \sqrt{16} = \text{normal} / 4$
- `linewidth.THin`: $\text{normal} / \sqrt{8} = \text{normal} / 2\sqrt{2}$
- `linewidth.Thin`: $\text{normal} / \sqrt{4} = \text{normal} / 2$
- `linewidth.thin`: $\text{normal} / \sqrt{2}$
- `linewidth.normal`: $0.02 * \text{unit.w_cm}$ (default)
- `linewidth.thick`: $\text{normal} \cdot \sqrt{2}$
- `linewidth.Thick`: $\text{normal} \cdot \sqrt{4} = \text{normal} \cdot 2$
- `linewidth.THick`: $\text{normal} \cdot \sqrt{8} = \text{normal} \cdot 2\sqrt{2}$
- `linewidth.THICK`: $\text{normal} \cdot \sqrt{16} = \text{normal} \cdot 4$
- `linewidth.THICK`: $\text{normal} \cdot \sqrt{32} = \text{normal} \cdot 4\sqrt{2}$
- `linewidth.THICK`: $\text{normal} \cdot \sqrt{64} = \text{normal} \cdot 8$
- `linewidth.clear`: Clears any preceding *linewidth* setting.

A visual chart of all predefined line caps, joins, miter limits, line styles, and line widths is given in *Appendix: Path styles in style module*.

2.5.2 Fill styles

All fill styles inherit from `fillstyle`:

class `style.fillrule(even_odd)`

Selects the rule used to determine the interior of self-intersecting or compound closed paths when filling:

- `fillrule.nonzero_winding` (`even_odd=0`, default): Uses the nonzero winding number rule.
- `fillrule.even_odd` (`even_odd=1`): Uses the alternating even-odd rule.
- `fillrule.clear`: Clears any preceding *fillrule* setting.

Colors (*Module color*), transparency (`color.transparency`), and patterns (*Module pattern*) inherit from both `strokestyle` and `fillstyle`.

2.6 Decorators: *deco*

Unlike styles, which only set graphics state parameters, path decorators depend on the concrete path to which they are applied: they specify whether the path is stroked or filled and can attach additional graphical ornaments (such as arrowheads, text labels, hatching lines, or braces) or exclude intervals of the path from stroking. All path decorators inherit from *deco*.

2.6.1 Stroking and filling

deco.stroked

A decorator that strokes the outline of the path. Calling `deco.stroked(styles=[])` returns a new stroke decorator with a list *styles* of `style.strokeStyle` instances (such as `style.linewidth`, `style.linestyle`, or `color.color`) applied locally only to the stroked outline, without affecting fills or ornaments on the same path.

- `deco.stroked` can be passed directly in an attribute list when no local stroke styles are needed.
- `deco.stroked.clear` removes any preceding `deco.stroked` decorator from an attribute list.

deco.filled

A decorator that fills the interior of the path. Calling `deco.filled(styles=[])` returns a new fill decorator with a list *styles* of `style.fillstyle` instances (such as `color.color`, `pattern.pattern`, or `style.fillrule`) applied locally only to the fill operation.

- `deco.filled` can be passed directly in an attribute list when no local fill styles are needed.
- `deco.filled.clear` removes any preceding `deco.filled` decorator from an attribute list.

2.6.2 Arrows

class `deco.arrow(attrs=[], pos=1, reversed=0, size=_base, angle=45, constriction=0.8)`

A decorator that attaches an arrowhead to a path. The outer contours of the arrowhead are constructed by cutting a segment of arc length *size* from the path itself and rotating it by $\pm\text{angle}/2$ around the tip, so arrowheads along curved paths naturally curve to follow the path's geometry.

Parameters:

- *attrs*: List of decorators (*deco*), fill styles (`style.fillstyle`), and stroke styles (`style.strokeStyle`) controlling how the arrowhead path is drawn. *attrs* is merged with the default attributes [`style.linestyle.solid`, `deco.filled`]. Thus, by default arrowheads are filled (and not stroked); passing `deco.stroked` in *attrs* also strokes the arrowhead outline (using a solid line style by default even if the main path is dashed), and passing [`deco.stroked`, `deco.filled.clear`] draws an unfilled, stroked arrowhead.
- *pos*: Relative arc-length position $\in [0, 1]$ along the path at which the arrowhead is placed (default 1, the end of the path; 0 is the beginning of the path).
- *reversed*: When 0 (default), the arrowhead points forward along the direction of the path; when 1, it points backward against the direction of the path. When an arrowhead sits strictly at the beginning (`pos=0`, `reversed=1`) or end (`pos=1`, `reversed=0`) of the path, the stroked main path is automatically shortened so that its line cap does not protrude through the sharp tip of the arrowhead.
- *size*: Arc length of the arrowhead sides (default `_base = 6 * unit.v_pt`).
- *angle*: Opening angle of the arrowhead in degrees (default 45).
- *constriction*: Ratio of the indented back-center distance to the arrowhead length (default 0.8). Setting `constriction=None` disables the indentation and yields a triangular arrowhead with a straight base.
- `arrow.clear`: Clears any preceding *arrow* decorator from an attribute list.

Calling an existing [arrow](#) instance with keyword or positional arguments `arrow(attrs=None, pos=None, reversed=None, size=None, angle=None, constriction=...)` returns a new [arrow](#) instance with the specified parameters overridden.

`deco.barrow`

`deco.earrow`

Predefined [arrow](#) instances for placing an arrowhead at the beginning of a path (`barrow = arrow(pos=0, reversed=1)`) or at the end of a path (`earrow = arrow()`). Both `barrow` and `earrow` can be used directly in an attribute list (e.g. `[deco.earrow]`), called to override parameters (e.g. `deco.earrow([color.rgb.red], angle=30)`), or cleared via `deco.barrow.clear` / `deco.earrow.clear` (which alias `arrow.clear`).

Both `barrow` and `earrow` provide 13 predefined size instances as attributes, forming a geometric progression with factor $\sqrt{2}$ around `_base = 6 * unit.v_pt`:

- `barrow.SMALL, earrow.SMALL`: size = `_base / sqrt(64)` (base/8)
- `barrow.SMALL, earrow.SMALL`: size = `_base / sqrt(32)` (base/4 $\sqrt{2}$)
- `barrow.SMALL, earrow.SMALL`: size = `_base / sqrt(16)` (base/4)
- `barrow.SMALL, earrow.SMALL`: size = `_base / sqrt(8)` (base/2 $\sqrt{2}$)
- `barrow.Small, earrow.Small`: size = `_base / sqrt(4)` (base/2)
- `barrow.small, earrow.small`: size = `_base / sqrt(2)`
- `barrow.normal, earrow.normal`: size = `_base` (6 * `unit.v_pt`)
- `barrow.large, earrow.large`: size = `_base * sqrt(2)`
- `barrow.Large, earrow.Large`: size = `_base * sqrt(4)` (base · 2)
- `barrow.LARGE, earrow.LARGE`: size = `_base * sqrt(8)` (base · 2 $\sqrt{2}$)
- `barrow.LARGE, earrow.LARGE`: size = `_base * sqrt(16)` (base · 4)
- `barrow.LARGE, earrow.LARGE`: size = `_base * sqrt(32)` (base · 4 $\sqrt{2}$)
- `barrow.LARGE, earrow.LARGE`: size = `_base * sqrt(64)` (base · 8)

Each predefined size instance is itself an [arrow](#) instance and can be called to customize its attributes, angle, or constriction—for example, `deco.earrow.Large([deco.filled([color.rgb.red]), style.linewidth.normal])` or `deco.earrow.normal(constriction=None)`. See [Appendix: Arrows in deco module](#) in the Appendix for a visual illustration.

2.6.3 Text decorators

class `deco.text`(*text*, *textattrs*=[], *angle*=0, *relangle*=None, *textdist*=0.2, *relarclenpos*=0.5, *arclenfrombegin*=None, *arclenfromend*=None, *texrunner*=None)

Typesets *text* and attaches the resulting text box at a point along the path:

- *text*: String to typeset.
- *textattrs*: List of text attributes merged with the defaults `[text.halign.center, text.vshift.mathaxis]`.
- *angle*: Absolute angle in degrees (measured counter-clockwise from the positive *x*-axis) specifying the direction in which the text box is displaced by *textdist* via `box.box.linealign()` (default 0).
- *relangle*: Angle in degrees relative to the path's local tangent direction at the anchor point. When not None, overrides *angle*.
- *textdist*: Distance between the anchor point on the path and the aligned text box (default 0.2).

- *relarclenpos*: Relative arc-length position $\in [0, 1]$ along the path (default `0.5`, the midpoint). Used when neither *arclenfrombegin* nor *arclenfromend* is given.
- *arclenfrombegin*: Arc length from the beginning of the path to the anchor point (mutually exclusive with *arclenfromend*).
- *arclenfromend*: Arc length from the end of the path to the anchor point (mutually exclusive with *arclenfrombegin*).
- *texrunner*: Optional text engine instance (defaults to the canvas's text engine).

class `deco.curvedtext`(*text*, *textattrs*=[], *relarclenpos*=0.5, *arclenfrombegin*=None, *arclenfromend*=None, *texrunner*=None, *exclude*=None)

Typesets *text* in `singlecharmode` so that each character is individually positioned and rotated to follow the local tangent of the curved path:

```
c.stroke(path.curve(0, 0, 1, 2, 3, -1, 4, 1),
         [deco.curvedtext("Text along a curve", [text.vshift.mathaxis], exclude=0.
↪1)])
```

- *text*: String to typeset along the path.
- *textattrs*: List of text attributes, transformations, or fill styles (merged with default [`text.halign.center`]).
- *relarclenpos*, *arclenfrombegin*, *arclenfromend*: Specify the anchor position along the path in the same way as in `deco.text`.
- *texrunner*: Optional text engine instance.
- *exclude*: When set to a PyX length (e.g. `0.1`), cuts out the stroked underlying path across the horizontal extent of each character enlarged on both sides by *exclude*.

2.6.4 Hatching, gradient, brace, and diagnostic decorators

class `deco.linehatched`(*dist*, *angle*, *strokestyles*=[], *cross*=0)

Fills the interior of a path by clipping a set of parallel stroked lines spaced by *dist* at angle *angle* (in degrees) to the path. Unlike PostScript/PDF tiling patterns (*pattern*), *linehatched* draws explicit vector lines clipped to the path and is therefore supported by all output backends and printers. It acts simultaneously as an exclusive *linehatched* attribute and clears `deco.filled`.

- *dist*: Spacing between adjacent hatch lines.
- *angle*: Angle of the hatch lines in degrees.
- *strokestyles*: List of `style.strokestyle` instances merged with [`style.linewidth.THIN`].
- *cross*: When true (1), draws a second orthogonal set of hatch lines at `angle + 90`.
- `linehatched.clear`: Clears any preceding *linehatched* decorator.

Predefined instances are provided for single hatching at 0° , 45° , 90° , 135° (`linehatched0`, `linehatched45`, `linehatched90`, `linehatched135`) and cross-hatching at 0° and 45° (`crosslinehatched0`, `crosslinehatched45`). Each predefined instance provides 13 spacing variants scaled by powers of $\sqrt{2}$ around `0.1 * unit.v_cm`: `.SMALL`, `.SMALL`, `.SMALL`, `.SMALL`, `.SMALL`, `.small`, `.normal`, `.large`, `.Large`, `.LARGE`, `.LARGE`, and `.LARGE`.

class `deco.colorgradient`(*grad*, *attrs*=[], *steps*=20)

Splits the path into *steps* segments of equal arc length and strokes each segment with the color sampled from the *color.gradient* *grad* merged with *attrs*.

```
class deco.brace(reverse=1, stretch=None, dist=None, fillattrs=[], totalheight=12 * unit.x_pt, barthickness=0.5
    * unit.x_pt, innerstrokesthickness=0.25 * unit.x_pt, outerstrokesthickness=0.25 * unit.x_pt,
    innerstrokesrelheight=0.6, outerstrokesrelheight=0.7, innerstrokesangle=30,
    outerstrokesangle=25, slantstrokesangle=5, innerstrokeessmoothness=2.0,
    outerstrokeessmoothness=2.5, middlerelpos=0.5)
```

Draws a curly brace between the start point and end point of a path (usually used with `canvas.canvas.draw()` so the connecting line itself is not stroked). Predefined brace instances are `leftbrace`, `rightbrace`, `belowbrace`, `abovebrace`, and `straightbrace`, and `brace.clear` removes any preceding `brace` decorator.

```
class deco.shownormpath(normline_attrs=[], normcurve_attrs=[], endpoint_size=0.05 * unit.v_cm,
    endpoint_attrs=[], controlline_attrs=[], controlpoint_size=0.05 * unit.v_cm,
    controlpoint_attrs=[])
```

A diagnostic decorator that visualizes the structure of a normalized path: straight segments (`normpath.normline_pt`) are stroked in blue by default, cubic Bézier segments (`normpath.normcurve_pt`) in green, Bézier control polygons with red dashed lines and filled red control points, and segment endpoints with filled black circles. Passing `None` for any of the `*_attrs` arguments disables drawing that element.

PATHS

Paths are the geometric backbone of PyX. A path describes a sequence of connected or disconnected line segments, circular or elliptical arcs, and cubic Bézier curves in the plane.

PyX provides two complementary representations of paths:

1. `path.path`: A PostScript-style path composed of arbitrary `pathitem` instances (*moveto*, *lineto*, *curveto*, *arc*, *closepath*, etc.). When written to PostScript or SVG, these path items map directly to native PostScript operators or SVG path commands.
2. `normpath.normpath`: A *normalized path* consisting of a list of `normpath.normsubpath` objects, which in turn contain only straight lines (`normpath.normline_pt`) and cubic Bézier curves (`normpath.normcurve_pt`) in absolute point coordinates, subject to a numerical accuracy *epsilon* (default 10^{-5} pt). All advanced geometric operations—such as computing arc lengths, finding intersections, splitting paths, or evaluating tangents and curvatures—are performed on normalized paths.

Because `path.path` automatically delegates geometric queries to its cached `path.normpath()` representation, you can call methods like `path.intersect()`, `path.at()`, `path.split()`, or `path.tangent()` directly on any `path.path` instance.

3.1 Constructing paths

A `path` can be constructed either by passing `pathitem` instances to `path` or by using one of the predefined shape classes such as *line*, *rect*, *circle*, *ellipse*, or *curve*.

For example, to construct a triangle with a rounded top corner and intersect it with a circle:

```
from pyx import *

c = canvas.canvas()
triangle = path.path(path.moveto(0, 0),
                     path.lineto(4, 0),
                     path.arct(2, 3, 0, 0, 0.5),
                     path.closepath())
circle = path.circle(2, 1, 1.5)

c.stroke(triangle)
c.stroke(circle, [color.rgb.blue])

# Mark all intersection points with small red circles
for p in triangle.intersect(circle)[0]:
    x, y = triangle.at(p)
    c.fill(path.circle(x, y, 0.08), [color.rgb.red])
```

(continues on next page)

```
c.writePDFfile("intersect")
```

Two paths can be combined in two different ways:

- **Addition** ($p = p1 + p2$ or $p1 += p2$): Concatenates the subpaths of $p1$ and $p2$ into a single compound path without connecting the end of $p1$ to the start of $p2$.
- **Joining** ($p = p1 << p2$ or $p = p1.joined(p2)$ or $p1.join(p2)$): Connects the end of the last open subpath of $p1$ to the beginning of the first open subpath of $p2$ (inserting a straight line segment if the two endpoints do not coincide) so that they form a single continuous subpath.

3.2 Path items

Every element passed to `path` must be an instance of a subclass of `pathitem`. Each coordinate-accepting class has both a standard version (taking coordinates in PyX user lengths or explicit `unit` lengths) and a `_pt` version (taking coordinates in PostScript points).

3.2.1 Subpath control

```
class path.moveto(x, y)
```

```
class path.moveto_pt(x_pt, y_pt)
```

Starts a new subpath and sets the current point to (x, y) .

```
class path.rmoveto(dx, dy)
```

```
class path.rmoveto_pt(dx_pt, dy_pt)
```

Starts a new subpath and moves the current point by $(\Delta x, \Delta y)$ relative to the previous current point.

```
class path.closepath
```

Closes the current subpath by connecting the current point back to the starting point of the subpath with a straight line segment.

3.2.2 Straight lines

```
class path.lineto(x, y)
```

```
class path.lineto_pt(x_pt, y_pt)
```

Appends a straight line segment from the current point to (x, y) , which becomes the new current point.

```
class path.rlineto(dx, dy)
```

```
class path.rlineto_pt(dx_pt, dy_pt)
```

Appends a straight line segment from the current point (x_0, y_0) to $(x_0 + \Delta x, y_0 + \Delta y)$, which becomes the new current point.

```
class path.multilineto_pt(points_pt)
```

Appends straight line segments from the current point through a sequence of coordinate pairs $[(x1_pt, y1_pt), (x2_pt, y2_pt), \dots]$ in points. This class is provided for performance when constructing polylines with many vertices.

3.2.3 Circular arcs

```
class path.arc(x, y, r, angle1, angle2)
```

class `path.arc_pt(x_pt, y_pt, r_pt, angle1, angle2)`

Appends a counterclockwise circular arc centered at (x, y) with radius r from $angle1$ to $angle2$ (in degrees). If the path already has a current point in an open subpath, a straight line segment is first added from the current point to the starting point of the arc; otherwise, the arc starts a new subpath. After the arc is added, the current point is the endpoint of the arc at $angle2$.

class `path.arcn(x, y, r, angle1, angle2)`

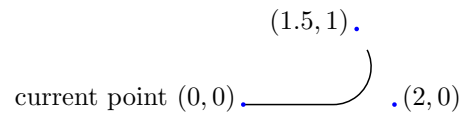
class `path.arcn_pt(x_pt, y_pt, r_pt, angle1, angle2)`

Same as `arc`, except that the arc is drawn in the clockwise direction from $angle1$ down to $angle2$.

class `path.arct(x1, y1, x2, y2, r)`

class `path.arct_pt(x1_pt, y1_pt, x2_pt, y2_pt, r_pt)`

Appends a circular arc of radius r that is tangent both to the line from the current point (x_0, y_0) to (x_1, y_1) and to the line from (x_1, y_1) to (x_2, y_2) . A straight line segment is first drawn from (x_0, y_0) to the first point of tangency $(x_{t,1}, y_{t,1})$, followed by the tangent arc ending at the second point of tangency $(x_{t,2}, y_{t,2})$, which becomes the new current point:



3.2.4 Bézier curves

class `path.curveto(x1, y1, x2, y2, x3, y3)`

class `path.curveto_pt(x1_pt, y1_pt, x2_pt, y2_pt, x3_pt, y3_pt)`

Appends a cubic Bézier curve from the current point (x_0, y_0) to (x_3, y_3) with first control point (x_1, y_1) and second control point (x_2, y_2) . The parametric curve is given by

$$B(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3, \quad t \in [0, 1].$$

After the curve is added, (x_3, y_3) becomes the new current point.

class `path.rcurveto(dx1, dy1, dx2, dy2, dx3, dy3)`

class `path.rcurveto_pt(dx1_pt, dy1_pt, dx2_pt, dy2_pt, dx3_pt, dy3_pt)`

Appends a cubic Bézier curve whose control points and endpoint are specified relative to the current point (x_0, y_0) before the operation: $(x_0 + \Delta x_1, y_0 + \Delta y_1)$, $(x_0 + \Delta x_2, y_0 + \Delta y_2)$, and $(x_0 + \Delta x_3, y_0 + \Delta y_3)$.

class `path.multicurveto_pt(points_pt)`

Appends multiple cubic Bézier curves from a sequence of 6-tuples $[(x1_pt, y1_pt, x2_pt, y2_pt, x3_pt, y3_pt), \dots]$ in points, provided for performance when constructing paths with many curve segments.

3.3 Class `path`

class `path.path(*pathitems)`

Constructs a path from the given *pathitems*.

A `path` instance supports sequence indexing (`p[i]` returns the i -th `pathitem`), `len(p)` (number of path items), `p1 + p2` and `p1 += p2` (concatenating subpaths), and `p1 << p2` (joining paths via `joined()`).

Note

In the methods below, any parameter *params* that specifies a location (or a list/tuple of locations) along the path accepts either:

- a *normpath.normpathparam* instance (as returned by *begin()*, *end()*, *arclentoparam()*, or *intersect()*), or
- a PyX length (or number in user units, or in points for *_pt* methods), which is automatically interpreted as the arc length measured from the beginning of the path!

Furthermore, methods decorated to accept *params* or *lengths* (such as *at()*, *arclentoparam()*, *paramtoarclen()*, *curvature_pt()*, *rotation()*, *tangent()*, and *trafo()*) return a single value when passed a single parameter, and a list of values when passed a sequence of parameters.

append(*apathitem*)

Appends a single *pathitem* to the path in place.

extend(*pathitems*)

Appends an iterable of *pathitem* instances to the path in place.

arclen()

arclen_pt()

Returns the total arc length of the path as a PyX length (or in points).

arclentoparam(*lengths*)

arclentoparam_pt(*lengths_pt*)

Returns the *normpath.normpathparam* instance(s) corresponding to the given arc length(s).

paramtoarclen(*params*)

paramtoarclen_pt(*params*)

Returns the arc length(s) (as PyX lengths or in points) corresponding to the given *normpath.normpathparam* instance(s).

at(*params*)

at_pt(*params*)

Returns the coordinate tuple (*x*, *y*) of PyX lengths (or (*x_pt*, *y_pt*) in points) at the specified parameter(s) or arc length(s).

atbegin()

atbegin_pt()

Returns the coordinate tuple (*x*, *y*) (or (*x_pt*, *y_pt*)) of the start point of the first subpath.

atend()

atend_pt()

Returns the coordinate tuple (*x*, *y*) (or (*x_pt*, *y_pt*)) of the end point of the last subpath.

begin()

Returns the *normpath.normpathparam* corresponding to the beginning of the path.

end()

Returns the *normpath.normpathparam* corresponding to the end of the path.

bbox()

Returns the exact *bbox.bbox* enclosing the path.

curvature_pt(*params*)

Returns the signed curvature in pt^{-1} at the specified parameter(s) or arc length(s) in points. The curvature is positive when the path turns to the left (counterclockwise) and negative when it turns to the right.

intersect(*other*)

Intersects the path with *other* (another *path* or *normpath.normpath*) and returns a 2-tuple (*params_self*, *params_other*) of lists of *normpath.normpathparam* instances for the intersection points along *self* and *other*, sorted along *self*.

join(*other*)

Joins *other* to the end of *self* in place and returns *self*.

joined(*other*)

Returns a new path consisting of *self* and *other* joined together (also available via the << operator: *self* << *other*).

normpath(*epsilon*=...)

Converts the path into a *normpath.normpath* instance. When *epsilon* is omitted, the default normalized path is cached on the path instance.

path()

Returns *self* (provided so that *path* and *normpath.normpath* can be used interchangeably).

reversed()

Returns a new *normpath.normpath* with the orientation of all subpaths reversed.

rotation(*params*)**rotation_pt**(*params*)

Returns a *trafo.trafo* rotation at *params* (or arc lengths) that rotates the positive *x*-axis onto the unit tangent vector of the path and the positive *y*-axis onto the left-hand normal vector.

split(*params*)**split_pt**(*params*)

Splits the normalized path at the given parameter(s) or arc length(s) and returns a list of *normpath.normpath* segments. Note that even when a single split point is passed, a list of two path segments is returned.

tangent(*params*, *length*=1)**tangent_pt**(*params*, *length_pt*)

Returns a *normpath.normpath* line segment of length *length* (default 1 user unit) starting at the point *params* and pointing in the tangent direction of the path.

trafo(*params*)**trafo_pt**(*params*)

Returns a *trafo.trafo* transformation at *params* (or arc lengths) that maps the origin (0,0) to the point on the path and rotates the coordinate axes so that the local *x*-axis points along the path tangent and the local *y*-axis points along the left-hand normal.

transformed(*trafo*)

Returns a new *normpath.normpath* transformed according to the affine transformation *trafo*.

3.4 Normalized paths: *normpath*

While casual users rarely need to instantiate classes from *normpath* directly, PyX developers and advanced users working with intersections, path splitting, or custom decorators and deformaters interact closely with *normpath*, *normsubpath*, and *normpathparam*.

3.4.1 Why normalized paths?

PostScript paths can contain arcs, relative displacements, and unclosed or implicitly connected subpaths. Evaluating intersections or arc-length integrals directly on such heterogeneous items would be error-prone and slow. PyX therefore normalizes any `path.path` into a `normpath`:

- A `normpath` is a list of `normsubpath` instances.
- Each `normsubpath` is a contiguous open or closed chain of `normsubpathitem` instances—exclusively `normline_pt` (straight lines) and `normcurve_pt` (cubic Bézier curves)—together with a precision length `epsilon` (default 10^{-5} pt).
- Line or curve segments shorter than `epsilon` are accumulated as a `skippedline` rather than creating degenerate zero-length items with undefined tangents, and are merged into the adjacent segment once the accumulated displacement exceeds `epsilon`.

`normpath.set(epsilon=None)`

Sets the global default accuracy `epsilon` in points (initially $1e-5$) used when constructing new `normsubpath` instances.

3.4.2 Class `normpathparam`

class `normpath.normpathparam(normpath, normsubpathindex, normsubpathparam)`

Identifies a point along `normpath`. Here `normsubpathindex` is the integer index of the `normsubpath` within `normpath`, and `normsubpathparam` is a floating-point parameter t along that subpath, where the integer part $\lfloor t \rfloor$ selects the `normsubpathitem` and the fractional part $t - \lfloor t \rfloor \in [0, 1]$ is the parameter on that line or Bézier curve.

`normpathparam` instances support rich arithmetic and comparisons via arc length:

- Adding or subtracting a PyX length (e.g. `p + 0.5*unit.t_cm` or `p - 1`) moves along the path by that arc length and returns the resulting `normpathparam`!
- Adding or subtracting two `normpathparam` instances belonging to the same `normpath`, or multiplying/dividing a `normpathparam` by a scalar factor, operates on their arc lengths and returns the corresponding `normpathparam`.
- Comparing two `normpathparam` instances (`p1 < p2`, `p1 == p2`) compares their positions (`normsubpathindex`, `normsubpathparam`) along the path; comparing a `normpathparam` with a length compares its arc length from the start of the path with that length.

`arclen()`

`arclen_pt()`

Returns the arc length from the beginning of the `normpath` up to this parameter position as a PyX length (or in points).

3.4.3 Class `normpath`

class `normpath.normpath(normsubpaths=None)`

Constructs a normalized path from a list of `normsubpath` instances (or an empty list if `None`).

`normpath` provides all the geometric and manipulation methods of `path.path` (`path.path.append()`, `path.path.extend()`, `path.path.arclen()`, `path.path.arclen_pt()`, `path.path.arclentoparam()`, `path.path.arclentoparam_pt()`, `path.path.at()`, `path.path.at_pt()`, `path.path.atbegin()`, `path.path.atbegin_pt()`, `path.path.atend()`, `path.path.atend_pt()`, `path.path.bbox()`, `path.path.begin()`, `path.path.end()`, `path.path.curvature_pt()`, `path.path.intersect()`, `path.path.join()`, `path.path.joined()`, `path.path.normpath()`, `path.path.paramtoarclen()`,

`path.path.paramtoarclen_pt()`, `path.path.reversed()`, `path.path.rotation()`, `path.path.rotation_pt()`, `path.path.split()`, `path.path.split_pt()`, `path.path.tangent()`, `path.path.tangent_pt()`, `path.path.trafo()`, `path.path.trafo_pt()`, and `path.path.transformed()`, plus:

copy()

Returns a deep copy of the *normpath* and all its subpaths.

path()

Converts the *normpath* back into an equivalent *path.path* consisting of *path.moveto_pt*, *path.lineto_pt*, *path.curveto_pt*, and *path.closepath* items.

Indexing `np[i]` returns the *i*-th *normsubpath*, and `len(np)` returns the number of subpaths.

3.4.4 Class *normsubpath* and subpath items

class `normpath.normsubpath(normsubpathitems=[], closed=0, epsilon=_marker)`

Represents a single connected open or closed normalized subpath.

- *normsubpathitems*: List of *normline_pt* and *normcurve_pt* instances. Adjacent items must share endpoints within *epsilon*.
- *closed*: Boolean or integer flag indicating whether the subpath is closed.
- *epsilon*: Precision threshold in points (defaults to the module-level `normpath._epsilon = 1e-5`, or `None` to disable filtering of short segments).

Key attributes and methods:

normsubpathitems

The list of *normline_pt* and *normcurve_pt* items in the subpath.

closed

Whether the subpath is closed.

epsilon

Accuracy threshold in points for this subpath.

append(anormsubpathitem)

extend(normsubpathitems)

Appends one or more *normsubpathitem* instances to an open subpath, automatically coalescing segments shorter than *epsilon*.

close()

Closes the subpath in place, appending a closing *normline_pt* back to the start point if necessary.

segments(params)

Takes a sequence of at least two subpath parameter floats and returns a list of *normsubpath* instances for the segments between consecutive parameters.

pathitems()

Returns the list of *path.pathitem* instances representing this subpath.

class `normpath.normline_pt(x0_pt, y0_pt, x1_pt, y1_pt)`

A normalized straight line segment from (x0_pt, y0_pt) to (x1_pt, y1_pt) in points.

class `normpath.normcurve_pt(x0_pt, y0_pt, x1_pt, y1_pt, x2_pt, y2_pt, x3_pt, y3_pt)`

A normalized cubic Bézier curve segment with control points (x0_pt, y0_pt) through (x3_pt, y3_pt) in points.

3.5 Predefined shapes

For convenience, `path` provides subclasses of `path` for common geometric primitives, each in a user-length version and a `_pt` version:

class `path.line`(*x1*, *y1*, *x2*, *y2*)

class `path.line_pt`(*x1_pt*, *y1_pt*, *x2_pt*, *y2_pt*)

A straight line segment from (x_1, y_1) to (x_2, y_2) .

class `path.curve`(*x0*, *y0*, *x1*, *y1*, *x2*, *y2*, *x3*, *y3*)

class `path.curve_pt`(*x0_pt*, *y0_pt*, *x1_pt*, *y1_pt*, *x2_pt*, *y2_pt*, *x3_pt*, *y3_pt*)

A single cubic Bézier curve from (x_0, y_0) to (x_3, y_3) with control points (x_1, y_1) and (x_2, y_2) .

class `path.rect`(*x*, *y*, *width*, *height*)

class `path.rect_pt`(*x_pt*, *y_pt*, *width_pt*, *height_pt*)

A closed axis-aligned rectangle with lower-left corner (x, y) , *width*, and *height*.

class `path.circle`(*x*, *y*, *radius*, *arcepsilon*=0.1)

class `path.circle_pt`(*x_pt*, *y_pt*, *radius_pt*, *arcepsilon*=0.1)

A closed circle with center (x, y) and *radius* (constructed from a counterclockwise arc from *arcepsilon* to $360 - \text{arcepsilon}$ degrees closed with `closepath`).

class `path.ellipse`(*x*, *y*, *a*, *b*, *angle*, ***kwargs*)

class `path.ellipse_pt`(*x_pt*, *y_pt*, *a_pt*, *b_pt*, *angle*, ***kwargs*)

A closed ellipse centered at (x, y) with semi-axes *a* and *b*, where the first axis *a* is rotated counterclockwise by *angle* degrees relative to the positive *x*-axis.

METAPOST-LIKE PATHS

In addition to constructing paths with explicit Bézier control points via `path`, PyX provides the `metapost.path` module, which implements the spline-choosing algorithm of John Hobby and Donald Knuth from MetaPost. Instead of specifying control points manually, you specify a sequence of *knots* (points through which the curve must pass, optionally with tangent angles or endpoint curvatures) separated by *links* (such as smooth curves with adjustable tension or straight lines). PyX then solves the global system of equations to determine cubic Bézier control points that maintain continuous tangents and “mock” curvatures at all interior smooth knots.

The resulting `metapost.path.path` is a subclass of `normpath.normpath`, so it supports all standard PyX path and normalized path operations (such as stroking, filling, intersecting, splitting, and deforming).

4.1 Example

To draw a closed smooth curve passing through four points, and an open curve with prescribed start and end tangent angles:

```
from pyx import *
from pyx.metapost.path import beginknot, endknot, smoothknot, roughknot, curve, line

c = canvas.canvas()

# Closed smooth loop through three knots
closed_loop = metapost.path.path([
    smoothknot(0, 0), curve(),
    smoothknot(2, 1), curve(),
    smoothknot(1, 2.5), curve()
])
c.stroke(closed_loop, [color.rgb.blue])

# Open curve starting at 45 degrees and ending with a straight line segment
open_curve = metapost.path.path([
    beginknot(3, 0, angle=45), curve(ltension=1.2),
    smoothknot(4.5, 1.5), line(keepangles=True),
    endknot(6, 0.5)
])
c.stroke(open_curve, [color.rgb.red])

c.writePDFfile("metapost")
```

4.2 Path construction: *path*

class `metapost.path.path(elems, epsilon=None)`

Constructs a *normpath.normpath* from a list *elems* of alternating knots and links.

- If *elems* begins with a *beginknot* and ends with an *endknot*, an open path is constructed.
- If *elems* contains only interior knots (*smoothknot* or *roughknot*) and links, the last link connects the last knot back to the first knot, forming a closed path.
- Multiple open subpaths can be constructed within a single *path* by placing an *endknot* followed immediately by a new *beginknot* (without a link between them).
- *epsilon* specifies the numerical precision in points used for the resulting *normpath.normsubpath* (defaulting to the module-level precision set via *set()*).

`metapost.path.set(epsilon=None)`

Sets the module-level default precision *epsilon* in points (initially 1e-5).

4.3 Knots

Knots define the points through which the path passes and specify boundary or continuity conditions for the incoming (left) and outgoing (right) curve segments. Every knot class is available both in a user-length version (where angles are given in **degrees**) and a *_pt* version (where coordinates are given in points and angles in **radians**).

class `metapost.path.beginknot(x, y, curl=1, angle=None)`

class `metapost.path.beginknot_pt(x_pt, y_pt, curl=1, angle=None)`

class `metapost.path.startknot(x, y, curl=1, angle=None)`

Starts an open subpath at (x, y) . You may specify either the relative endpoint curvature *curl* (default 1, meaning the curvature at the endpoint matches that of the adjacent knot) or an explicit outgoing tangent direction *angle* (in degrees for *beginknot*, or radians for *beginknot_pt*). *startknot* is an alias for *beginknot*.

class `metapost.path.endknot(x, y, curl=1, angle=None)`

class `metapost.path.endknot_pt(x_pt, y_pt, curl=1, angle=None)`

Ends an open subpath at (x, y) . You may specify either the relative endpoint curvature *curl* (default 1) or an explicit incoming tangent direction *angle* (in degrees for *endknot*, or radians for *endknot_pt*).

class `metapost.path.smoothknot(x, y)`

class `metapost.path.smoothknot_pt(x_pt, y_pt)`

class `metapost.path.knot(x, y)`

An interior knot at (x, y) where both the tangent direction and the mock curvature are continuous across the knot. *knot* is an alias for *smoothknot*.

class `metapost.path.roughknot(x, y, lcurl=1, rcurl=None, langle=None, range=None)`

class `metapost.path.roughknot_pt(x_pt, y_pt, lcurl=1, rcurl=None, langle=None, range=None)`

An interior knot at (x, y) where the curve may have a corner or prescribed left/right curls or tangent angles.

- On the incoming (left) side, specify either *lcurl* (default 1) or *langle* (incoming tangent angle).
- On the outgoing (right) side, specify either *rcurl* or *range* (outgoing tangent angle). If neither *rcurl* nor *range* is given, the right-side condition defaults to the same type and value as the left-side condition (so *roughknot*(*x*, *y*, *langle*=30) produces a smooth tangent at 30 degrees, whereas *roughknot*(*x*, *y*, *langle*=30, *range*=90) produces a sharp corner).

4.4 Links

Links are placed between consecutive knots in the list passed to `path` and determine how the two knots are connected.

class `metapost.path.tensioncurve`(*ltension=1*, *latleast=False*, *rtension=None*, *ratleast=None*)

class `metapost.path.curve`(*ltension=1*, *latleast=False*, *rtension=None*, *ratleast=None*)

Connects two knots with a cubic Bézier spline segment determined by the MetaPost solver.

- *ltension*: Tension at the starting (left) knot of the segment (default 1; must be at least 0.75). Higher tension pulls the curve tighter toward a straight line between the knots.
- *latleast*: When `True`, corresponds to MetaPost's `tension atleast` directive at the starting knot, which increases the tension automatically if needed to avoid inflection points near the knot.
- *rtension*: Tension at the ending (right) knot of the segment (defaults to *ltension* when `None`).
- *ratleast*: When `True`, applies `tension atleast` at the ending knot (defaults to *latleast* when `None`).

`curve` is an alias for `tensioncurve`.

class `metapost.path.controlcurve`(*lcontrol*, *rcontrol*)

class `metapost.path.controlcurve_pt`(*lcontrol_pt*, *rcontrol_pt*)

Connects two knots with a cubic Bézier curve whose two control points are set explicitly to the coordinate 2-tuples *lcontrol* and *rcontrol* (in user coordinates or points). Adjacent smooth curves on either side of those knots will adapt their tangents and curvatures to match this explicit segment.

class `metapost.path.line`(*keepangles=False*)

Connects two knots with a straight line segment.

- When *keepangles* is `False` (the default), the straight line acts as an endpoint boundary for adjacent curves (equivalent to `--` in MetaPost), allowing corners at the knots.
- When *keepangles* is `True`, the tangent directions of the curves entering the left knot and leaving the right knot are constrained to match the angle of the straight line segment (equivalent to `---` in MetaPost), ensuring a smooth tangent transition into and out of the straight segment.

DEFORMERS

Deformers take an existing path and replace it with a geometrically modified *normpath.normpath*—for example, rounding sharp corners, constructing an offset curve at a constant parallel distance, or wrapping a spring-like cycloid along the path.

All deformers inherit from `baseclasses.deformer` and can be used in two ways:

1. **As drawing attributes** passed to `canvas.canvas.stroke()`, `canvas.canvas.fill()`, or `canvas.canvas.draw()`. When drawing a path, PyX applies all deformers in the attribute list in order before applying decorators and styles:

```
from pyx import *

c = canvas.canvas()
p = path.rect(0, 0, 3, 2)
c.stroke(p, [style.linestyle.dotted])
c.stroke(p, [deformer.smoothed(0.5), color.rgb.red])
c.writePDFfile("smoothed")
```

2. **Directly as path transformations** by calling the deformer's `deform()` method on a path or normalized path to obtain the deformed *normpath.normpath* for further geometric operations (such as intersections or splitting):

```
rounded_rect = deformer.smoothed(0.5).deform(path.rect(0, 0, 3, 2))
```

Every deformer instance is also callable: calling an existing instance with keyword arguments returns a new deformer with those parameters updated. Each deformer class also provides a `.clear` attribute (e.g. `deformer.smoothed.clear`, `deformer.cycloid.clear`) to remove preceding instances of that deformer from an attribute list.

5.1 Corner smoothing: *smoothed*

class `deformer.smoothed(radius, softness=1, obeycurv=0, relskipthres=0.01)`

class `deformer.cornersmoothed(radius, softness=1, obeycurv=0, relskipthres=0.01)`

Replaces corners between consecutive segments of a path with smooth cubic Bézier curves. Within each subpath item, the path is cut back at arc length *radius* from the corner (or at the midpoint of the segment if the segment is shorter than $2 * radius$), and the removed corner is bridged as follows:

- **Between two straight lines:** Two optimized cubic Bézier curves are inserted whose curvature vanishes at the transition points to the straight lines and whose curvature derivative vanishes at the midpoint of the corner. The shape is controlled by *softness* $\in (0, 1]$.
- **When at least one adjacent segment is curved:** A single cubic Bézier curve is inserted that matches the tangents and curvatures of the adjacent segments at the cut points.

Parameters:

- *radius*: Arc-length distance from each corner at which the original path is cut and replaced by the smoothing curve(s).
- *softness*: Factor in $(0, 1]$ controlling how tightly corners between straight lines are rounded (default 1). Smaller values produce sharper bends closer to the original corner.
- *obeycurv*: When 0 (the default), only the magnitude of the curvature is matched at curved segments while its sign is chosen to avoid cusps or loops; when 1, the signed curvature of the adjacent curve is strictly preserved.
- *relskipthres*: Segments shorter than $\text{relskipthres} * \text{radius}$ are skipped during smoothing (default 0.01).

cornersmoothed is the canonical class name; *smoothed* is provided as a convenient alias.

deform(*basepath*)

Returns a new *normpath.normpath* with all corners of *basepath* smoothed.

5.2 Parallel offset curves: *parallel*

class `deformer.parallel`(*distance*, *relerr*=0.05, *sharpoutercorners*=False, *dointersection*=True, *checkdistanceparams*=[0.5], *lookforcurvatures*=11, *searchstep*=0.01, *debug*=None)

Constructs a parallel offset curve at a constant normal *distance* from the original path.

A positive *distance* offsets the curve to the **left** of the original path (relative to its parameterization direction); a negative *distance* offsets it to the **right**. For a counter-clockwise closed path (such as *path.rect* or *path.circle*), a negative *distance* therefore expands the shape outward, while a positive *distance* shrinks it inward.

The parallel path is constructed in two stages:

1. For each segment of the normalized path, an offset line or cubic Bézier approximation is built. At convex corners, either a circular arc of radius `abs(distance)` is inserted around the corner (default) or the offset segments are extended to a sharp corner (*sharpoutercorners*=True). Regions where the original curve bends more tightly than $1 / \text{distance}$ toward the offset side are split at points of infinite curvature.
2. When *dointersection* is True (the default), self-intersections and loops in the offset path are detected and trimmed away so that only the true envelope at distance `abs(distance)` from the original path remains.

Parameters:

- *distance*: Signed normal offset distance (PyX length).
- *relerr*: Maximum allowed relative error $\text{relerr} * \text{abs}(\text{distance})$ when approximating offset Bézier curves (default 0.05).
- *sharpoutercorners*: When True, outer (convex) corners are joined with sharp mitered lines instead of circular arcs (default False).
- *dointersection*: When True (default), self-intersecting loops in the offset curve are removed. Set to False to keep the raw untrimmed offset curve.
- *checkdistanceparams*: List of parameter values in $(0, 1)$ at which the normal distance between the original curve and its cubic Bézier approximation is verified before subdividing (default [0.5]).
- *lookforcurvatures*: Number of sample points per curve segment used to locate critical curvature crossings $\kappa = 1/\text{distance}$ (default 11).

deform(*basepath*)

Returns the offset *normpath.normpath* at *distance* from *basepath*.

5.3 Spring / coil paths: *cycloid*

```
class deformer.cycloid(radius=0.5 * unit.t_cm, halfloops=10, skipfirst=1 * unit.t_cm, skiplast=1 * unit.t_cm,
                        curvesperhloop=3, sign=1, turnangle=45)
```

Wraps a cycloid (the 2D projection of a 3D helical spring) along the path, using the original path as the central axis of the coil. This deformer is commonly used in physics and engineering diagrams to draw springs, inductors, or gluons.

Parameters:

- *radius*: Radius of the cycloid loops perpendicular to the base path (default 0.5 * unit.t_cm).
- *halfloops*: Total number of half-loops along the deformed portion of the path (default 10).
- *skipfirst*: Arc length at the beginning of each subpath left undeformed as a lead-in segment (default 1 * unit.t_cm).
- *skiplast*: Arc length at the end of each subpath left undeformed as a lead-out segment (default 1 * unit.t_cm).
- *curvesperhloop*: Number of cubic Bézier segments used to approximate each half-loop (default 3).
- *sign*: 1 (default) to start the first half-loop to the left of the path, or -1 to start to the right.
- *turnangle*: Viewing angle in degrees of the 3D helix projection (default 45). Setting *turnangle*=0 produces a sine-like wave without loops; increasing *turnangle* toward 90 produces tightly overlapping circular coils.

deform(basepath)

Returns a new *normpath.normpath* with the cycloid wrapped along each subpath of *basepath* (subpaths shorter than *skipfirst* + *skiplast* + 2 * *radius* * sin(*turnangle*) are left unchanged with a warning).

CANVAS

A *canvas* is the central workspace in PyX. It acts both as a drawing surface—providing methods to stroke, fill, and draw paths and to typeset text—and as a hierarchical container that groups canvas items (paths, text boxes, sub-canvases, bitmaps, and embedded EPS, PDF, or SVG files) with shared visual attributes, transformations, clipping paths, and drawing layers.

6.1 Working with canvases

6.1.1 Drawing and grouping

Every drawing operation on a canvas (*canvas.stroke()*, *canvas.fill()*, *canvas.draw()*, *canvas.text()*) creates a *baseclasses.canvasitem* and inserts it into the canvas via *canvas.insert()*, returning the inserted item.

Because a *canvas* is itself a *baseclasses.canvasitem*, you can create sub-canvases, draw on them in local coordinates, and insert them into a parent canvas with transformations or styles applied to the entire group:

```
from pyx import *

# Create a reusable sub-canvas in local coordinates
badge = canvas.canvas()
badge.fill(path.circle(0, 0, 0.5), [color.rgb.red])
badge.text(0, 0, "1", [text.halign.center, text.valign.middle, color.gray.white])

# Insert the sub-canvas multiple times at different positions and scales
c = canvas.canvas()
c.insert(badge, [trafo.translate(1, 1)])
c.insert(badge, [trafo.scale(1.5), trafo.translate(3, 1)])

c.writePDFfile("badges")
```

6.1.2 Layers

By default, items on a canvas are drawn in the exact order in which they are inserted (later items paint over earlier items). When constructing complex figures, it is often convenient to organize drawing operations into named *layers* (such as "background", "fill", "main", "labels") whose back-to-front order is independent of when individual items are added.

The *canvas.layer()* method creates or retrieves a sub-canvas associated with a layer name and allows you to position it relative to existing layers using the *above* and *below* parameters:

```
c = canvas.canvas()
bg = c.layer("background")
```

(continues on next page)

(continued from previous page)

```
fg = c.layer("foreground", above="background")

# Even if we draw on fg first and bg second, bg is rendered behind fg
fg.text(1, 1, "Foreground text", [text.halign.center, text.valign.middle])
bg.fill(path.rect(0, 0.5, 2, 1), [color.rgb.yellow])
```

6.1.3 Clipping

A canvas can be clipped to the interior of an arbitrary closed *path.path* by passing a *clip* instance in the canvas constructor's *attrs* list:

```
clipped = canvas.canvas([canvas.clip(path.circle(1, 1, 1))])
clipped.fill(path.rect(0, 0, 2, 2), [color.rgb.blue])

c = canvas.canvas()
c.insert(clipped)
```

6.2 Class *canvas*

class `canvas.canvas(attrs=[], textengine=None)`

Constructs a new canvas.

- *attrs*: Optional list of canvas-wide attributes applied to all contents of the canvas. Allowed attributes are `style.strokestyle`, `style.fillstyle` (including `color.color` and *pattern.pattern*), `trafo.trafo_pt`, and *clip*.
- *textengine*: Optional text engine instance (such as a custom *text.TextEngine*, *text.LatexEngine*, or *text.UnicodeEngine*) used by *text()* and *text_pt()* on this canvas. When None (the default), `text.defaulttextengine` is used at the time *text()* is called.

items

List of `baseclasses.canvasitem` objects contained in the canvas.

layers

Dictionary mapping layer names to their corresponding sub-canvas instances.

trafo

The cumulative `trafo.trafo_pt` transformation applied to the canvas.

bbox()

Returns the enclosing *bbox.bbox* of all items in the canvas (taking canvas transformations and any *clip* path into account). Note that for stroked paths, the bounding box reflects the geometric path rather than the stroke width.

draw(path, attrs)

Draws *path* on the canvas according to the attributes in *attrs*, which may contain `baseclasses.deformer`, `baseclasses.deco`, `style.strokestyle`, `style.fillstyle`, and `trafo.trafo_pt` instances. Returns the inserted `deco.decoratedpath` canvas item.

stroke(path, attrs=[])

Strokes *path* on the canvas; equivalent to `self.draw(path, [deco.stroked] + list(attrs))`. Returns the inserted `deco.decoratedpath` canvas item.

fill(*path*, *attrs*=[])

Fills *path* on the canvas; equivalent to `self.draw(path, [deco.filled] + list(attrs))`. Returns the inserted `deco.decoratedpath` canvas item.

insert(*item*, *attrs*=[])

Inserts *item* (an instance of `baseclasses.canvasitem`, such as another `canvas`, a `text.textbox_pt`, a `bitmap.bitmap`, an `epsfile.epsfile`, a `pdffile.pdffile`, or an `svgfile.svgfile`) into the canvas. If *attrs* is non-empty, *item* is wrapped in a sub-canvas with *attrs* applied. Returns *item*.

layer(*name*, *above*=None, *below*=None)

Creates or retrieves the layer sub-canvas identified by the string *name*. Dotted names (such as "axes.ticks") create nested sub-layers inside the parent layer "axes".

- *above*: Name of an existing layer immediately above which the layer should be positioned.
- *below*: Name of an existing layer immediately below which the layer should be positioned (cannot be combined with *above*).

Returns the `canvas` instance representing the layer.

text(*x*, *y*, *expr*, *textattrs*=[], ***kwargs*)

text_pt(*x_pt*, *y_pt*, *expr*, *textattrs*=[], ***kwargs*)

Typesets *expr* at (*x*, *y*) (in PyX user lengths or points) using the canvas's text engine, inserts the resulting text box into the canvas, and returns it. See `text.SingleEngine.text_pt()` for details on *textattrs* and *kwargs*.

settextengine(*textengine*)

Sets the text engine (`text.TextEngine`, `text.LatexEngine`, or `text.UnicodeEngine`) to be used by subsequent `text()` and `text_pt()` calls on this canvas.

6.2.1 Output and Ghostscript conversion methods

For single-page figures, `canvas` provides shortcut methods that wrap the canvas in a `document.page` and a single-page `document.document` and write the output directly. Any keyword arguments accepted by `document.page` (such as `paperformat`, `rotated`, `centered`, `fittosize`, `margin`, `bboxenlarge`, `bbox`) as well as any keyword arguments accepted by the respective writer (such as `title`, `stripfonts`, `textaspath`, `writebbox`; see *Documents*) can be passed directly to these methods:

`canvas.writeEPSfile(file=None, **kwargs)`

`canvas.writePSfile(file=None, **kwargs)`

`canvas.writePDFfile(file=None, **kwargs)`

`canvas.writeSVGfile(file=None, **kwargs)`

Writes the canvas to an EPS, PostScript, PDF, or SVG file. *file* can be a filename string (the extension `.eps`, `.ps`, `.pdf`, or `.svg` is appended automatically if omitted), a binary writable stream, "-" to write to `sys.stdout.buffer`, or None to derive the filename automatically from `sys.argv[0]`.

`canvas.writetofile(filename, **kwargs)`

Determines the output format from the extension of *filename* (`.eps`, `.ps`, `.pdf`, or `.svg`) and writes the canvas accordingly.

`canvas.pipeGS(filename=None, device='png16m', resolution=100, gs='gs', gsoptions=[],
textalphabits=4, graphicsalphabits=4, ciecolor=False, input='eps', **kwargs)`

Renders the canvas using Ghostscript (gs) into any output format supported by Ghostscript (such as PNG or JPEG).

- *filename*: Output filename. If *filename* does not end with a file extension and *device* starts with "png" or "jpeg", the extension .png or .jpg is appended automatically. If *filename* is `None`, the base name is derived from `sys.argv[0]`. If *filename* is "-", Ghostscript writes to standard output and `pipeGS()` returns the resulting output as a bytes object!
- *device*: Ghostscript output device (default "png16m" for 24-bit RGB PNG; use e.g. "pngalpha" for PNG with transparent background or "jpeg" for JPEG).
- *resolution*: Rendering resolution in DPI (default 100).
- *gs*: Ghostscript executable name (default "gs").
- *gsoptions*: Additional command-line option strings passed to Ghostscript.
- *textalphabits*, *graphicsalphabits*: Antialiasing subsampling factors (default 4; pass `None` to omit).
- *ciecolor*: When `True`, passes `-dUseCIEColor` to Ghostscript.
- *input*: Format piped from PyX into Ghostscript: "eps" (default) or "pdf".
- *kwargs*: Additional keyword arguments forwarded to `document.page` and the EPS/PDF writer.

6.3 Class *clip*

class `canvas.clip(path)`

Constructs a clipping attribute from the closed `path.path` *path*. When passed in the *attrs* list of `canvas`, all drawing inside that canvas is restricted to the interior of *path*.

DOCUMENTS

While a `canvas.canvas` represents an unbounded 2D drawing surface, the `document` module places one or more canvases onto physical pages with optional paper formats, margins, centering, rotation, and scaling, and writes the resulting single- or multi-page document to PostScript, EPS, PDF, or SVG files.

7.1 Creating single- and multi-page documents

For a single-page figure without specific page layout options, you can call `canvas.canvas.writeEPSfile()`, `canvas.canvas.writePDFfile()`, or `canvas.canvas.writeSVGfile()` directly on a canvas. Internally, those methods create a `page` and a `document` for you.

To create a multi-page PostScript or PDF document, or to control page layout explicitly, wrap each `canvas.canvas` in a `page` and collect them in a `document`:

```
from pyx import *

d = document.document()

for i in range(1, 4):
    c = canvas.canvas()
    c.text(0, 0, f"Page {i}", [text.halign.center])
    d.append(document.page(c, paperformat=document.paperformat.A4, centered=1))

d.writePSfile("multipage")
d.writePDFfile("multipage")
```

7.2 Class `page`

```
class document.page(canvas, pagename=None, paperformat=None, rotated=0, centered=1, fittosize=0,
                    margin=1 * unit.t_cm, bboxenlarge=1 * unit.t_pt, bbox=None)
```

Wraps a `canvas.canvas` as a document page and configures its placement and bounding box.

Parameters:

- `canvas`: The `canvas.canvas` containing the page graphics.
- `pagename`: Optional string naming the page (written in the `%%Page:` DSC comment in PostScript output; defaults to the 1-based page number).
- `paperformat`: Optional `paperformat` instance (such as `document.paperformat.A4` or `document.paperformat.Letter`) defining the physical dimensions of the page.

- *rotated*: When set to 1 (or True), rotates the page content by 90 degrees counterclockwise (landscape orientation) on the specified *paperformat* (default 0).
- *centered*: When set to 1 (default) and a *paperformat* is specified, centers the canvas bounding box on the paper.
- *fittosize*: When set to 1 and a *paperformat* is specified, scales the canvas uniformly so that its bounding box fits within the paper dimensions minus *margin* on all four sides (default 0).
- *margin*: Margin around the page content used when *fittosize* is enabled (default `1 * unit.t_cm`).
- *bboxenlarge*: Amount by which the automatically computed canvas bounding box is enlarged on all four sides to account for finite line widths at the edges of the figure (default `1 * unit.t_pt`).
- *bbox*: Explicit `bbox.bbox` overriding the canvas's automatically computed bounding box (when specified, *bboxenlarge* is not applied).

7.3 Class *paperformat*

class `document.paperformat`(*width*, *height*, *name=None*)

Defines a paper size of the given *width* and *height* (PyX lengths) and optional PostScript media *name*.

Predefined standard paper formats available as attributes of *paperformat* are:

7.4 Class *document* and output writers

class `document.document`(*pages=None*)

Represents a sequence of *page* instances.

- *pages*: Optional initial list of *page* instances (defaults to an empty list).

append(*page*)

Appends a *page* instance to the document.

writeEPSfile(*file=None*, *title=None*, *stripfonts=True*, *textaspath=False*, *meshasbitmap=False*,
meshasbitmapresolution=300)

Writes a single-page document as an Encapsulated PostScript (EPS) file. Raises `ValueError` if the document does not contain exactly one page.

writePSfile(*file=None*, *writebbox=False*, *title=None*, *stripfonts=True*, *textaspath=False*,
meshasbitmap=False, *meshasbitmapresolution=300*)

Writes the document as a PostScript (PS) file.

writePDFfile(*file=None*, *title=None*, *author=None*, *subject=None*, *keywords=None*, *fullscreen=False*,
writebbox=False, *compress=True*, *compresslevel=6*, *stripfonts=True*, *textaspath=False*,
meshasbitmap=False, *meshasbitmapresolution=300*)

Writes the document as a PDF file.

writeSVGfile(*file=None*, *textaspath=True*, *meshasbitmapresolution=300*)

Writes a single-page document as an SVG 1.1 file. Raises `ValueError` if the document does not contain exactly one page.

writetofile(*filename*, ***kwargs*)

Determines the output writer from the extension of *filename* (`.eps`, `.ps`, `.pdf`, or `.svg`) and writes the document with the given *kwargs*.

7.4.1 Writer options

In all output methods above, *file* may be:

- a filename string (the appropriate file extension `.eps`, `.ps`, `.pdf`, or `.svg` is appended automatically if not already present),
- an open binary writable stream (such as an `io.BytesIO` instance),
- `"-"` to write the binary output to standard output (`sys.stdout.buffer`), or
- `None` to derive the base filename automatically from the running script name `sys.argv[0]` (replacing `.py` with the appropriate extension).

Note

For reproducible builds, both the PostScript and PDF writers honor the `SOURCE_DATE_EPOCH` environment variable when writing creation timestamps.

The keyword arguments controlling the output writers are:

- *stripfonts* (bool, default `True`; EPS, PS, PDF): When `True`, embedded Type 1 and OpenType fonts are subsetting so that only the glyphs actually used in the document are included, substantially reducing file size.
- *textaspath* (bool, default `False` for EPS/PS/PDF and `True` for SVG): When `True`, all text glyphs are converted into vector outline paths instead of embedding font resources. In SVG output, this defaults to `True` so that SVG files render identically in all browsers and vector editors without requiring SVG font support.
- *meshasbitmap* (bool, default `False`; EPS, PS, PDF): When `True`, Gouraud-shaded triangular meshes (mesh, used e.g. by [graph.style.surface](#)) are rasterized into a bitmap image instead of using native PostScript Level 3 / PDF shading objects. This avoids rendering artifacts and large file sizes in some viewers and printers. (In SVG output, meshes are always rendered as bitmaps.)
- *meshasbitmapresolution* (int, default `300`): Resolution in DPI used when rasterizing meshes into bitmaps.
- *writebbox* (bool, default `False`; PS, PDF): In PostScript output, writes `%%BoundingBox` and `%%PageBoundingBox` comments even when a paper format is used. In PDF output, writes a `/CropBox` entry alongside the `/MediaBox` when a paper format is used.
- *title* (str or `None`; EPS, PS, PDF): Document title metadata.
- *author*, *subject*, *keywords* (str or `None`; PDF): Document metadata strings written to the PDF `/Info` dictionary.
- *fullscreen* (bool, default `False`; PDF): Requests that the PDF viewer open the document in full-screen presentation mode (`/PageMode /FullScreen`).
- *compress* (bool, default `True`; PDF) and *compresslevel* (int, default `6`; PDF): Controls `zlib`/Flate compression of PDF content and font streams.

8.1 Overview and Quick Start

The `text` module provides PyX's text typesetting facilities. Depending on your needs and environment, PyX offers two complementary text rendering engines:

- **TeX and LaTeX engines** (*TexEngine* and *LatexEngine*): The default typesetting backend. PyX runs Donald E. Knuth's TeX¹ or LaTeX as a background process, giving you full access to TeX's mathematical typesetting, macro expansion, kerning, ligatures, and LaTeX packages.
- **Direct Unicode font engine** (*UnicodeEngine*): A lightweight, pure-Python text engine that renders Unicode strings directly using Type 1 or OpenType/TrueType (`.otf/.ttf`) fonts without invoking an external TeX process.

8.1.1 Basic Usage

In most scripts, you do not create a text engine directly. Instead, you call `canvas.canvas.text()` (which delegates to the module-level default engine) and configure the global engine via `text.set()` and `text.preamble()`:

```
from pyx import *

# Switch the default engine from plain TeX to LaTeX and load packages
text.set(cls=text.LatexEngine)
text.preamble(r"\usepackage{amsmath}")

c = canvas.canvas()
c.text(0, 0, r"$\int_{-\infty}^{\infty} e^{-x^2}\,dx = \sqrt{\pi}$",
      [text.halign.center, text.vshift.mathaxis, text.size.Large])
c.writePDFfile("gauss")
```

Calling `text.text()` (or `canvas.canvas.text()`) returns a `texttextbox_pt` (or `textbox`) instance—a canvas-like object that knows its exact typographic width, height, and depth, and can be aligned, transformed, or queried for its bounding box before or after insertion into a canvas.

8.2 Architecture of the TeX Interface

Understanding how PyX communicates with TeX/LaTeX is helpful when customizing preambles, handling TeX warnings, or optimizing scripts that typeset many text fragments.

1. **Direct TeX execution:** PyX does not parse or restrict TeX syntax itself; instead, it passes your text directly to an external `tex` or `latex` process wrapped in a thin TeX macro for box measurement and shipout. Because arbitrary

¹ <https://en.wikipedia.org/wiki/TeX>

TeX primitives can read files on disk, untrusted user input should only be typeset inside a restricted environment (see *Typesetting Untrusted Text*).

2. **Persistent background process:** As soon as the first preamble command or `text()` call is issued, PyX launches the TeX interpreter as a child process bound to a *SingleEngine* instance. This single process stays alive across subsequent `text()` calls so that TeX only initializes once and macro definitions persist.
3. **Immediate box metrics and output parsing:** For each text fragment, PyX places the typeset material into a TeX box, instructs TeX to print the box's exact width, height, and depth to stdout followed by a unique end-of-command marker, and ships the box out to its own page in the DVI² file. PyX immediately parses stdout using a chain of *texmessage* parsers:
 - Expected status messages are stripped silently.
 - Overfull/underfull box notices and font warnings are logged via the "pyx" logger at WARNING level.
 - Any unrecognized output left over after all *texmessage* parsers have run raises a *TexResultError* with a detailed diagnostic report.

Because TeX reports box dimensions immediately on stdout, PyX knows the exact dimensions of every text box (width, height, depth) right away—even before reading the DVI file.

4. **DVI reading and font embedding:** When final output (EPS, PS, PDF, or SVG) is written—or when a coordinate marker inside the text box is queried via `texttextbox_pt.marker()`—PyX reads the corresponding page from the DVI file, resolves virtual fonts (.vfb), TeX font metrics (.tfm), and font map files (psfonts.map / pdftex.map), and embeds the required Type 1 font outlines directly into the output document.

8.3 TeX and LaTeX Engines

PyX separates the low-level management of a single TeX process (*SingleEngine*) from the high-level engine interface (*MultiEngine*) that automatically restarts TeX if needed (for instance, when the DVI file must be flushed and read mid-script without *TeX IPC Mode*).

8.3.1 Single-Process Engines

```
class text.SingleEngine(cmd, texenc='ascii', usefiles=[], texipc=config.getboolean('text', 'texipc', 0),
                        copyinput=None, dvitype=False, errordetail=errordetail.default,
                        texmessages_start=[], texmessages_end=[], texmessages_preamble=[],
                        texmessages_run=[])
```

Base class for the TeX interface.

Note

This class cannot be used directly. It is the base class for all tex engines and provides most of the implementation. Still, to the end user the parameters except for *cmd* are important, as they are preserved in derived classes usually.

Parameters

- **cmd** (*list of str*) – command and arguments to start the TeX interpreter
- **texenc** (*str*) – encoding to use in the communication with the TeX interpreter
- **usefiles** (*list of str*) – list of supplementary files to be copied to and from the temporary working directory (see *Debugging TeX Typesetting Issues* for usage details)

² https://en.wikipedia.org/wiki/Device_independent_file_format

- **texipc** (*bool*) – *TeX IPC Mode* flag.
- **copyinput** (*None or str or file*) – filename or file to be used to store a copy of all the input passed to the TeX interpreter
- **dvitype** (*bool*) – flag to turn on dvitype-like output
- **errordetail** (*errordetail*) – verbosity of the *TexResultError*
- **texmessages_start** (list of *texmessage* parsers) – additional message parsers at interpreter startup
- **texmessages_end** (list of *texmessage* parsers) – additional message parsers at interpreter shutdown
- **texmessages_preamble** (list of *texmessage* parsers) – additional message parsers for preamble output
- **texmessages_run** (list of *texmessage* parsers) – additional message parsers for typeset output

texmessages_start_default = [*<function texmessage.start>*]

default *texmessage* parsers at interpreter startup

texmessages_end_default = [*<function texmessage.end>*, *<function texmessage.font_warning>*, *<function texmessage.rerun_warning>*, *<function texmessage.nobbl_warning>*]

default *texmessage* parsers at interpreter shutdown

texmessages_preamble_default = [*<function texmessage.load>*]

default *texmessage* parsers for preamble output

texmessages_run_default = [*<function texmessage.font_warning>*, *<function texmessage.box_warning>*, *<function texmessage.package_warning>*, *<function texmessage.load_def>*, *<function texmessage.load_graphics>*]

default *texmessage* parsers for typeset output

preamble(*expr*, *texmessages*=[*]*)

Execute a preamble.

Parameters

- **expr** (*str*) – expression to be executed
- **texmessages** (list of *texmessage* parsers) – additional message parsers

Preambles must not generate output, but are used to load files, perform settings, define macros, *etc.* In LaTeX mode, preambles are executed before `\begin{document}`. The method can be called multiple times, but only prior to *SingleEngine.text()* and *SingleEngine.text_pt()*.

text_pt(*x_pt*, *y_pt*, *expr*, *textattrs*=[*]*, *texmessages*=[*]*, *fontmap*=*None*, *singlecharmode*=*False*)

Typeset text.

Parameters

- **x_pt** (*float*) – x position in pts
- **y_pt** (*float*) – y position in pts
- **expr** (*str* or *MultiEngineText*) – text to be typeset
- **textattrs** (list of *textattr*, *:class:* `trafo.trafo_pt`, and *style.fillstyle*) – styles and attributes to be applied to the text

- **texmessages** (list of [texmessage](#) parsers) – additional message parsers
- **fontmap** (*None* or *fontmap*) – force a fontmap to be used (instead of the default depending on the output format)
- **singlecharmode** (*bool*) – position each character separately

Returns

text output insertable into a canvas.

Return type

[texttextbox_pt](#)

Raises

TexDoneError: when the TeX interpreter has been terminated already.

text(*x*, *y*, **args*, ***kwargs*)

Typeset text.

This method is identical to [text_pt\(\)](#) with the only difference of using PyX lengths to position the output.

Parameters

- **x** (*PyX length*) – x position
- **y** (*PyX length*) – y position

class `text.SingleTexEngine`(*cmd=config.getlist('text', 'tex', ['tex']), lfs='10pt', **kwargs*)

Plain TeX interface.

This class adjusts the [SingleEngine](#) to use plain TeX.

Parameters

- **cmd** (*list of str*) – command and arguments to start the TeX interpreter
- **lfs** (*str or None*) – resemble LaTeX font settings within plain TeX by loading a lfs-file
- **kwargs** – additional arguments passed to [SingleEngine](#)

An lfs-file is a file defining a set of font commands like `\normalsize` by font selection commands in plain TeX. Several of those files resembling standard settings of LaTeX are distributed along with PyX in the `pyx/data/lfs` directory. This directory also contains a LaTeX file to create lfs files for different settings (LaTeX class, class options, and style files).

class `text.SingleLatexEngine`(*cmd=config.getlist('text', 'latex', ['latex']), docclass='article', docopt=None, pyxgraphics=True, texmessages_docclass=[], texmessages_begindoc=[], **kwargs*)

LaTeX interface.

This class adjusts the [SingleEngine](#) to use LaTeX.

Parameters

- **cmd** (*list of str*) – command and arguments to start the TeX interpreter in LaTeX mode
- **docclass** (*str*) – document class
- **docopt** (*str or None*) – document loading options
- **pyxgraphics** (*bool*) – activate graphics bundle support, see [Using the LaTeX graphics Bundle with PyX](#)
- **texmessages_docclass** (list of [texmessage](#) parsers) – additional message parsers at LaTeX class loading

- **texmessages_begindoc** (list of *texmessage* parsers) – additional message parsers at `\begin{document}`
- **kwargs** – additional arguments passed to *SingleEngine*

texmessages_docclass_default = [`<function texmessage.load>`]

default *texmessage* parsers at LaTeX class loading

texmessages_begindoc_default = [`<function texmessage.load>`, `<function texmessage.no_aux>`]

default *texmessage* parsers at `\begin{document}`

8.3.2 Multi-Run Wrapper Engines

MultiEngine and its subclasses (*TexEngine* and *LatexEngine*) wrap one or more *SingleEngine* instances. They provide the same public methods (*preamble()*, *text()*, and *text_pt()*), while automatically starting a fresh *SingleEngine* (and replaying all accumulated preamble commands) whenever the previous process had to be closed to read its DVI file.

class `text.MultiEngine(cls, *args, **kwargs)`

A restartable *SingleEngine* class

Parameters

- **cls** (*SingleEngine* class) – the class being wrapped
- **args** (*list*) – args at class instantiation
- **kwargs** (*dict*) – keyword args at at class instantiation

preamble(*expr*, *texmessages*=[])

resembles *SingleEngine.preamble()*

text_pt(*args, **kwargs)

resembles *SingleEngine.text_pt()*

text(*args, **kwargs)

resembles *SingleEngine.text()*

reset(*reinit*=False)

Start a new *SingleEngine* instance

Parameters

- **reinit** (*bool*) – replay *preamble()* calls on the new instance

After executing this function further preamble calls are allowed, whereas once a text output has been created, *preamble()* calls are forbidden.

class `text.TexEngine(*args, **kwargs)`

A restartable *SingleTexEngine* class

Parameters

- **args** (*list*) – args at class instantiation
- **kwargs** (*dict*) – keyword args at at class instantiation

class `text.LatexEngine(*args, **kwargs)`

A restartable *SingleLatexEngine* class

Parameters

- **args** (*list*) – args at class instantiation
- **kwargs** (*dict*) – keyword args at at class instantiation

8.3.3 Text Boxes and Coordinate Markers

Every call to `text()` or `text_pt()` on a TeX/LaTeX engine returns a `texttextbox_pt` instance:

```
class text.texttextbox_pt(x_pt, y_pt, left_pt, right_pt, height_pt, depth_pt, do_finish, fontmap,  
                        singlecharmode, fillstyles)
```

Text output.

An instance of this class contains the text output generated by PyX. It is a `baseclasses.canvasitem` and thus can be inserted into a canvas.

left

left extent of the text (PyX length)

right

right extent of the text (PyX length)

width

width of the text (PyX length)

height

height of the text (PyX length)

depth

height of the text (PyX length)

marker(*name*)

Return the position of a marker.

Parameters

name (*str*) – name of the marker

Returns

position of the marker

Return type

tuple of two PyX lengths

This method returns the position of the marker of the given name within, previously defined by the `\PyXMarker{name}` macro in the typeset text. Please do not use the `@` character within your name to prevent name clashes with PyX internal uses (although we don't the marker feature internally right now).

Similar to generating actual output, the marker function accesses the DVI output, stopping. The *TeX IPC Mode* feature will allow for this access without stopping the TeX interpreter.

Tip

Using PyX Markers inside TeX: You can record the exact position of any point inside a TeX expression by inserting `\PyXMarker{id}` in your TeX string and later calling `tb.marker("id")` on the returned text box to obtain its (*x*, *y*) coordinates on the canvas:

```
tb = c.text(0, 0, r"$y = \PyXMarker{rhs}\underbrace{a x^2 + b x + c}$")  
mx, my = tb.marker("rhs")
```

8.4 Module-Level Functions

For convenience, the `text` module maintains a default module-level text engine (`defaulttextengine`, which defaults to a `TexEngine` instance) and exposes its methods as module-level functions. Calling `text.set()` replaces `defaulttextengine` with a newly configured engine.

`text.defaulttextengine`

the current `MultiEngine` instance for the module level functions

`text.preamble`

`defaulttextengine.preamble` (bound method)

`text.text_pt`

`defaulttextengine.text_pt` (bound method)

`text.text`

`defaulttextengine.text` (bound method)

`text.reset`

`defaulttextengine.reset` (bound method)

`text.set(engine=None, cls=None, mode=None, *args, **kwargs)`

Setup a new module level `MultiEngine`

Parameters

- **engine** – the module level engine object to be used, i.e. `TexEngine`, `LatexEngine`, or `UnicodeEngine`
- **cls** (*Engine object, not instance*) – identical to *engine*
 deprecated
 use the engine argument instead
- **mode** (*str or None*) – "tex" for `TexEngine` or "latex" for `LatexEngine` with arbitrary capitalization
 deprecated
 use the engine argument instead
- **args** (*list*) – args at class instantiation
- **kwargs** (*dict*) – keyword args at at class instantiation

`text.escapestring(s, replace={" ": "~", "$": "\$", "&": "\&", " _": "\_", "%": "\%", "^": "\string^", "~": "\string~", "<": '\$<$}', ">": '\$>$}', "{": '\$\{ \$}', "}" : '\$ \} $}', "\": '\$ \setminus $}', "|": r"\$ \mid $}"))`

Escapes ASCII characters such that they can be typeset by TeX/LaTeX

8.5 TeX Output Parsers and Error Handling

When TeX or LaTeX runs, it writes status banners, file load notices, box warnings, and error messages to `stdout`. After stripping its own internal page and extent markers, PyX passes the remaining output string through a sequence of `texmessage` parser callables. Each parser inspects the output string and performs one of four actions:

1. **Unrelated output:** Return the string unchanged so subsequent parsers can inspect it.
2. **Expected/harmless message:** Strip the recognized message and return the remaining string.
3. **Warning:** Log a warning via the "pyx" logger (printed to `stderr` by default, e.g. for overfull/underfull boxes or font substitutions), strip the message, and return the remaining string.

4. **Error:** Raise `TexResultError` (or leave the error text in the returned string—any non-empty text remaining after all parsers have run automatically triggers a `TexResultError`).

The amount of context included in a `TexResultError` exception is controlled by `errordetail`:

class `text.errordetail`

Constants defining the verbosity of the `TexResultError`.

none = 0

Without any input and output.

default = 1

Input and parsed output shortened to 5 lines.

full = 2

Full input and unparsed as well as parsed output.

exception `text.TexResultError`

Error raised by `texmessage` parsers.

To accept unknown TeX output without raising an error, you can append `texmessage.warn` or `texmessage.ignore` to the `texmessages_*` lists. The built-in message parsers are:

class `text.texmessage`

Collection of TeX output parsers.

This class is not meant to be instantiated. Instead, it serves as a namespace for TeX output parsers, which are functions receiving a TeX output and returning parsed output.

In addition, this class also contains some generator functions (namely `texmessage.no_file` and `texmessage.pattern`), which return a function according to the given parameters. They are used to generate some of the parsers in this class and can be used to create others as well.

static `start(msg)`

Validate TeX/LaTeX startup message including scrollmode test.

Example:

```
>>> texmessage.start(r'''
... This is e-TeX (version)
... *! Undefined control sequence.
... <*> \raiseerror
...                               %
... ''')
'''
```

static `no_file(fileending, qualname=None)`

Generator function to ignore the missing file message for fileending.

static `no_aux(msg)`

Ignore the missing aux file message.

static `no_nav(msg)`

Ignore the missing nav file message.

static `end(msg)`

Validate TeX shutdown message.

static load(msg)

Ignore file loading messages.

Removes text starting with a round bracket followed by a filename ignoring all further text until the corresponding closing bracket. Quotes and/or line breaks in the filename are handled as needed for TeX output.

Without quoting the filename, the necessary removal of line breaks is not well defined and the different possibilities are tested to check whether one solution is ok. The last of the examples below checks this behavior.

Examples:

```
>>> texmessage.load(r''other (text.py) things'')
'other things'
>>> texmessage.load(r''other ("text.py") things'')
'other things'
>>> texmessage.load(r''other ("tex
... t.py" further (ignored)
... text) things'')
'other things'
>>> texmessage.load(r''other (t
... ext
... .py
... fur
... ther (ignored) text) things'')
'other things'
```

static load_def(msg)

Ignore font definition (*.fd and *.def) loading messages.

static load_graphics(msg)

Ignore graphics file (*.eps) loading messages.

static ignore(msg)

Ignore all messages.

Should be used as a last resort only. You should write a proper TeX output parser function for the output you observe.

static warn(msg)

Warn about all messages.

Similar to *ignore*, but writing a warning to the logger about the TeX output. This is considered to be better when you need to get it working quickly as you will still be prompted about the unresolved output, while the processing continues.

static pattern(p, warning, qualname=None)

Warn by regular expression pattern matching.

static box_warning(msg)

Warn about overfull/underfull box.

static font_warning(msg)

Warn about font substitutions of NFSS.

static package_warning(msg)

Warn about generic package messages.

static rerun_warning(msg)

Warn about rerun required message.

static nobbl_warning(msg)

Warn about no-bbl message.

8.6 Text Attributes (Alignment, Parboxes, Math Mode, Font Size)

Text attributes are passed in a list to `text.text()` or `canvas.canvas.text()` alongside transformations (`trafo.trafo`), colors (`color.color`), and decorators. They generate appropriate TeX/LaTeX wrapper fragments or geometric alignments and automatically resolve ordering dependencies.

8.6.1 Horizontal Alignment (*halign*)

class text.halign(boxhalign, flushhalign)

Controls both the horizontal alignment of the text box relative to its reference point (*x*, *y*) (*boxhalign*) and the horizontal alignment of text lines inside a multi-line *parbox* (*flushhalign*). Standard values are 0 (left), 0.5 (center), and 1 (right), though arbitrary floating point numbers are permitted.

The separate helper classes *boxhalign* and *flushhalign* allow setting box placement and line flushing independently. Predefined instances are provided as class attributes on *halign*:

boxleft

Left-aligns the text box at the reference point (*boxhalign*=0, leaves *flushhalign* unchanged).

boxcenter

Horizontally centers the text box at the reference point (*boxhalign*=0.5, leaves *flushhalign* unchanged).

boxright

Right-aligns the text box at the reference point (*boxhalign*=1, leaves *flushhalign* unchanged).

flushleft

raggedright

Left-aligns (ragged-right) lines inside a *parbox* (*flushhalign*=0, leaves *boxhalign* unchanged).

flushcenter

raggedcenter

Centers lines inside a *parbox* (*flushhalign*=0.5, leaves *boxhalign* unchanged).

flushright

raggedleft

Right-aligns (ragged-left) lines inside a *parbox* (*flushhalign*=1, leaves *boxhalign* unchanged).

left

Combines *boxleft* and *flushleft* (*halign*(0, 0)).

center

Combines *boxcenter* and *flushcenter* (*halign*(0.5, 0.5)).

right

Combines *boxright* and *flushright* (*halign*(1, 1)).

8.6.2 Vertical Alignment (*valign*) and Multi-Line Boxes (*parbox*)



Fig. 1: Vertical alignment of single-line text boxes using *valign* (left) and baseline selection in multi-line *parbox* boxes (right).

class `text.valign(valign)`

Sets the vertical alignment of the text box relative to its overall bounding height (from bottom of depth to top of height): 0 aligns at the top, 0.5 at the vertical center, and 1 at the bottom. See the left side of figure *Vertical alignment of single-line text boxes using valign (left) and baseline selection in multi-line parbox boxes (right)*.

top

Aligns the top of the text box with the reference point (`valign(0)`).

middle

Vertically centers the text box at the reference point (`valign(0.5)`).

bottom

Aligns the bottom of the text box with the reference point (`valign(1)`).

baseline

Clears any box-based vertical alignment (identical to `valign.clear`) so that the text box is aligned along its typographic baseline (the default).

class `text.parbox(width, baseline=top)`

Typesets text in a vertical paragraph box of fixed *width* (a PyX length or number in user units), enabling automatic line wrapping and explicit line breaks (`\`). By default, TeX typesets in LR (left-to-right) mode on a single line unless a *parbox* attribute is provided.

Because a multi-line paragraph has multiple baselines, *baseline* selects which baseline serves as the reference baseline of the box:

- `parbox.top` (default): the baseline of the first line.
- `parbox.middle`: the vertical center of the paragraph box.
- `parbox.bottom`: the baseline of the last line.

See the right side of figure *Vertical alignment of single-line text boxes using valign (left) and baseline selection in multi-line parbox boxes (right)*.

8.6.3 Vertical Shifts (*vshift*)

class `text.vshift(lowerratio, heightstr='0')`

Lowers the text box relative to its baseline by *lowerratio* times the height of the character(s) in *heightstr* (default "0"). Multiple *vshift* attributes accumulate; use `vshift.clear` to remove all vertical shifts.

bottomzero

`vshift(0)` — no vertical shift (aligns at the baseline of "0").

middlezero

`vshift(0.5)` — lowers the text by half the height of "0".

topzero

`vshift(1)` — lowers the text by the full height of "0".

mathaxis

Lowers the text by the height of TeX's mathematical axis (the vertical center of math operators such as minus signs and fraction bars). This is used by default for graph axis labels and titles.

8.6.4 Math Mode, Font Sizes, and Phantom Text

text.mathmode

Typesets the text in TeX display math mode (`\displaystyle ...`). Use `clearmathmode` to clear this attribute.

class text.size(*sizeindex=None, sizename=None, sizelist=defaultsizelist*)

Applies a TeX/LaTeX font size command from *sizelist* (default `["normalsize", "large", "Large", "LARGE", "huge", "Huge", None, "tiny", "scriptsize", "footnotesize", "small"]`). You can specify either an integer *sizeindex* into *sizelist* or a command name *sizename*.

Predefined instances corresponding to standard LaTeX font sizes are available as class attributes:

tiny

`size(-4)`

scriptsize

`size(-3)`

footnotesize

`size(-2)`

small

`size(-1)`

normalsize

`size(0)`

large

`size(1)`

Large

`size(2)`

LARGE

`size(3)`

huge

`size(4)`

Huge

`size(5)`

text.phantom

Typesets an invisible box that occupies the exact width, height, and depth of the given text without drawing any glyphs (equivalent to TeX's `\phantom`). Use `clearphantom` to clear this attribute.

8.7 UnicodeEngine: Direct Font Rendering without TeX

When you only need simple labels or want to use system OpenType/TrueType (.otf/.ttf) or Type 1 fonts without invoking an external TeX process, you can use *UnicodeEngine*.

```
from pyx import *

text.set(text.UnicodeEngine, size=12)

c = canvas.canvas()
c.text(0, 0, "Direct Unicode text rendering", [color.rgb.blue])
```

class `text.UnicodeEngine`(*font=_marker, size=10*)

A pure-Python text engine that renders Unicode strings directly using Type 1 (`font.T1font`) or OpenType/TrueType (`font.OTFfont`) fonts.

- *font*: Font instance (defaults to `T1font.from_fontname("cmr10")`). Other options would be `T1font.from_fontfile(filename)`, `OTFfont.from_fontname(fontname)`, and `OTFfont.from_fontfile(filename)`.
- *size*: Base font size in PostScript points (default 10).

text_pt(*x_pt, y_pt, text, textattrs=[], texmessages=[], fontmap=None, singlecharmode=False*)

text(*x, y, text, textattrs=[], **kwargs*)

Typesets *text* (which may be a string, *Text*, *StackedText*, *MultiEngineText*, or a list thereof) at (*x*, *y*) and returns a `unicodetextbox_pt` instance. Supports *mathmode*, *trafo.trafo_pt*, and *style*. fillstyle (such as `color.color`) attributes in *textattrs*.

class `text.Text`(*text, scale=1, shift=0*)

Wraps a string *text* for *UnicodeEngine* with a relative font *scale* factor and a vertical baseline *shift* (in units of the unscaled text size). This is useful for superscripts and subscripts when using *UnicodeEngine*, for example `["10", text.Text("-3", scale=0.7, shift=0.4)]`.

class `text.StackedText`(*texts, frac=False, align=0*)

Stacks a sequence of *texts* vertically for *UnicodeEngine* with horizontal alignment *align* (0 for left, 0.5 for center, 1 for right) and optional fraction bar *frac* (when `len(texts) == 2`).

class `text.MultiEngineText`(*tex, unicode*)

Holds two representations of a text fragment: *tex* (passed when a TeX/LaTeX engine is active) and *unicode* (passed when *UnicodeEngine* is active). PyX's axis texters use *MultiEngineText* so that exponential tick labels work seamlessly with both TeX engines and *UnicodeEngine*.

8.8 Using the LaTeX graphics Bundle with PyX

The packages in the standard LaTeX graphics bundle (`color`, `graphicx`, `rotating`, etc.) rely on driver-specific `\special` commands emitted into the DVI file. PyX ships with its own driver file `pyx.def` (located in `pyx/data/def/pyx.def`) that translates LaTeX `color`, `scaling`, `rotation`, and `EPS` inclusion commands into native PyX canvas operations.

By default, *SingleLatexEngine* enables `pyxgraphics=True`, which automatically injects `pyx.def` so you can write `\usepackage{color,graphicx}` in your LaTeX preamble without specifying a driver option. If you install `pyx.def` into your local TeX tree and configure `color.cfg` / `graphics.cfg`³, you can set `pyxgraphics=False` and load packages explicitly with `\usepackage[pyx]{color,graphicx}`.

³ Unless you specifically need custom driver configuration in your local TeX tree, leave `pyxgraphics=True` (the default) and omit the `[pyx]` package option in `\usepackage{color,graphicx}`.

8.8.1 Supported LaTeX Color Models and Specials

When you define or use colors in LaTeX via the `gray`, `cmymk`, `rgb`, `RGB`, or `hsb` models, or via named colors, PyX maps them directly to the corresponding *color* classes and predefined named colors (see appendix *Appendix: Named colors*). In addition, the `pyx` color model lets you pass any PyX color expression directly from LaTeX, such as `\textcolor{pyx}{color.cmyk.PineGreen}{colored text}`.

For developer reference, PyX's DVI reader interprets the following `\special` commands generated by `pyx.def`:

PyX:color_begin (model) (spec)

Pushes a color onto the graphics state. (*model*) is one of `gray`, `cmymk`, `rgb`, `hsb`, `texnamed`, or `pyxcolor`, and (*spec*) contains the model parameters or name.

PyX:color_end

Restores the previous color state.

PyX:epsinclude file= llx= lly= urx= ury= width= height= clip=0/1

Includes an Encapsulated PostScript (`.eps`) file scaled, positioned, and optionally clipped according to the specified bounding box coordinates.

PyX:scale_begin (x) (y) / PyX:scale_end

Starts and ends coordinate scaling relative to the current DVI reference point.

PyX:rotate_begin (angle) / PyX:rotate_end

Starts and ends coordinate rotation (in degrees) around the current DVI reference point.

8.9 Configuration and Debugging

As described in *Introduction*, PyX reads its system-wide defaults from `pyx/data/pyxrc`, followed by user-level settings in `~/ .pyxrc` (or `%APPDATA%\pyxrc` on Windows) and any file referenced by the `PYXRC` environment variable. Most options in `[general]` and `[text]` configure how PyX locates TeX files and runs the TeX/LaTeX interpreter.

8.9.1 TeX IPC Mode

To read the actual glyph positions or `\PyXMarker` coordinates of a text box, PyX must read the DVI page written by TeX. However, standard TeX buffers its DVI output until the TeX process terminates. When `texipc` is disabled (the default), accessing a marker or writing a canvas forces *MultiEngine* to close the running TeX process so the DVI file is flushed, and then spawn a new TeX process if further text is typeset afterward.

Most modern TeX distributions support the `--ipc` command-line option, which flushes the DVI file after every `\shipout`. Enabling `texipc = 1` in the `[text]` section of your `pyxrc` allows PyX to read DVI pages on the fly without terminating TeX, dramatically speeding up scripts that alternate between typesetting text and querying marker positions.

8.9.2 Debugging TeX Typesetting Issues

PyX provides several tools for diagnosing TeX errors and environment problems:

1. **Error verbosity** (*errordetail*): Pass `errordetail=text.errordetail.full` to `text.set()` to include the full TeX session transcript and input commands whenever a *TexResultError* is raised.
2. **Saving the TeX input stream** (*copyinput*): Pass a filename or writable file object as `copyinput="debug.tex"` to `text.set()` to record the exact TeX code PyX sends to the interpreter so you can run `tex` or `latex` on it manually.
3. **Preserving ``.log`` and ``.aux`` files** (*usefiles*): Each TeX process runs in a temporary working directory that is deleted on exit. Passing `usefiles=["myjob.log", "myjob.aux"]` to `text.set()` copies those files into and out of the temporary directory (matching PyX's internal `\jobname` while preserving the file extension), allowing you to inspect the full TeX `.log` file or share `.aux` files across runs.

4. Enabling ``INFO`` logging (`pyx.pyxinfo()`):

PyX logs all external command executions and file lookups to the standard Python logger "pyx" at INFO level. Calling `pyx.pyxinfo()` at the top of your script attaches a stream handler that prints these messages to `stderr`:

`pyx.pyxinfo()`

Make PyX a little verbose (for information or debugging)

This function enables info level on the "pyx" logger. It also adds some general information about the Python interpreter, the PyX installation, and the PyX configuration to the logger.

8.9.3 Typesetting Untrusted Text

Because TeX has primitives that can read arbitrary files accessible to the user running the script (and, in misconfigured environments, execute shell commands), never pass untrusted user input directly to TeX without isolation⁴. If you use PyX in a web service or multi-user environment, you can run TeX inside a `chroot` jail by configuring the `chroot` option, the TeX executable command, and the `filelocator` paths in `pyxrc`.

⁴ Although PyX wraps user text inside an internal TeX macro argument, catcode changes and macro expansion still allow arbitrary TeX primitives to be executed.

GRAPHS

9.1 Introduction

PyX provides a flexible, publication-quality plotting system for 1D, 2D (x - y), and 3D (x - y - z) data and mathematical functions. Its modular component architecture (described in section *Component Architecture and Canvas Layers*) cleanly separates graph geometry, data sources, plotting styles, legends (keys), and axes.

9.1.1 A Minimal Plot

Creating a graph in PyX takes three steps:

1. Instantiate a graph object such as `graph.graphxy`:

```
from pyx import *
g = graph.graphxy(width=8)
```

2. Call `plot()` with one or more data sources (and optional plot styles). For example, given a data file `graph.dat` containing two columns of numbers:

```
1  2
2  3
3  8
4 13
5 18
6 21
```

we can plot column 1 against column 2 with:

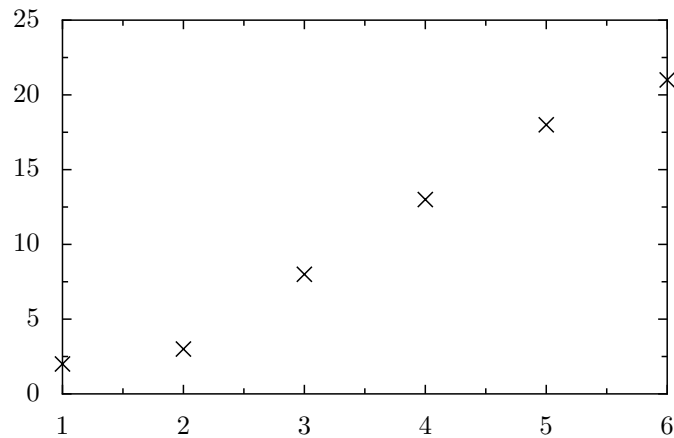
```
g.plot(graph.data.file("graph.dat", x=1, y=2))
```

When no explicit style list is passed to `plot()`, each data source uses its own default style—symbols (`graph.style.symbol`) for tabular data such as `graph.data.file`, and continuous lines (`graph.style.line`) for analytical functions such as `graph.data.function`.

3. Write the finished graph to a file. Because every graph instance is a subclass of `canvas.canvas`, you can call any canvas output method (or insert the graph into another canvas):

```
g.writeEPSfile("graph")
```

The resulting plot `graph.eps` is shown in figure *A minimal scatter plot of the data in graph.dat..*

Fig. 1: A minimal scatter plot of the data in `graph.dat`.

9.1.2 Combining Multiple Datasets and Functions

You can overlay multiple datasets and analytical functions on the same graph by calling `plot()` several times before writing the output. For example, to plot the parabola $y(x) = x^2$ alongside the data file:

```
g.plot(graph.data.function("y(x)=x**2"))
```

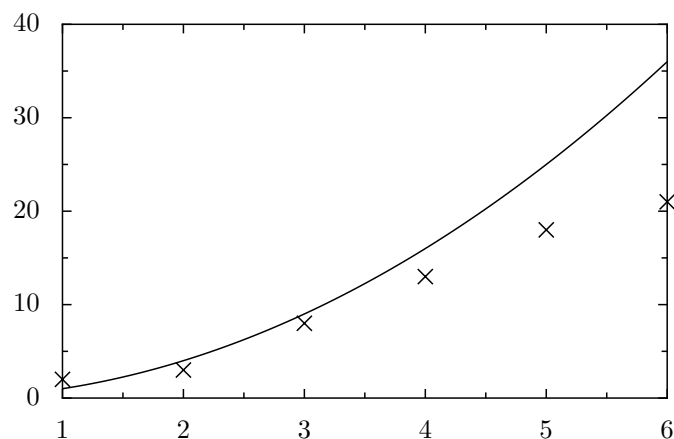


Fig. 2: Plotting tabular data from a file together with an analytical function.

9.1.3 Configuring Axes and Keyword-Splitting Shortcuts

In the examples above, the x and y axes were created automatically as linear axes (`graph.axis.linear`) and their ranges were determined from the plotted data. When plotting only an analytical function, or when you want fixed axis bounds, titles, or logarithmic scaling, you can pass explicit axis instances (see chapter [Axes](#)):

```
from pyx import *
g = graph.graphxy(width=8, x=graph.axis.linear(min=-5, max=5),
                  y=graph.axis.logarithmic())
g.plot(graph.data.function("y(x)=exp(x)"))
g.writePDFfile()
```

The resulting plot with a logarithmic y -axis is shown in figure *Plotting a function over a fixed x -range with a logarithmic y -axis.*

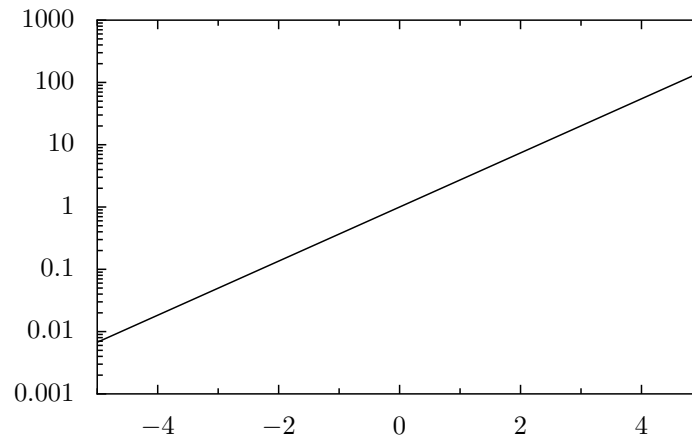


Fig. 3: Plotting a function over a fixed x -range with a logarithmic y -axis.

Tip

Concise axis configuration via underscore keyword splitting: Instead of instantiating `graph.axis.linear` or `graph.axis.painter.regular` explicitly just to set a few options, you can pass underscore-separated keyword arguments directly to `graph.graphxy` (powered by `utils.kwsplit()`). For example:

```
g = graph.graphxy(width=8,
                  x_min=0, x_max=10, x_title=r"$x$",
                  y_title=r"$f(x)$",
                  x_painter_gridattrs=[color.gray(0.8)])
```

Here `x_min`, `x_max`, and `x_title` are automatically forwarded to the `x` axis, and `x_painter_gridattrs` is forwarded to the painter of the `x` axis!

9.2 Component Architecture and Canvas Layers

PyX splits the work of constructing a graph across five cooperating modules:

graph (`graph.graph`)

Defines the overall geometry of the plot area using normalized *graph coordinates* $v \in [0, 1]$ along each dimension, manages the canvas layers, coordinates axis range adjustment and style selection, and provides geometric transformation methods between data coordinates, graph coordinates, and canvas coordinates.

data (`graph.data`)

Reads, generates, or transforms data columns (from files, Python lists, or mathematical expressions) and supplies points to the styles.

style (`graph.style`)

Translates data values into visual marks on the graph canvas (symbols, lines, error bars, histograms, bars, 3D surfaces, heatmaps/densities, arrows, and text labels).

key (`graph.key`)

Builds and positions the graph legend from the titles and style samples of the plotted items.

axis (`graph.axis`)

Maps data values to normalized graph coordinates $[0, 1]$ and partitions, formats, rates, and paints axis baselines, ticks, labels, titles, and grids (covered in detail in chapter [Axes](#)).

9.2.1 Canvas Layer Ordering in a Graph

Every `graph.graph` instance is a `canvas.canvas` pre-initialized with nine named layers in back-to-front drawing order:

1. "background" — graph background fill and border
2. "grid" — axis grid lines
3. "filldata" — filled data regions (such as bars, rectangles, and density bitmaps)
4. "axes.baseline" — axis baselines
5. "axes.ticks" — axis tick marks
6. "axes.labels" — axis tick labels
7. "axes.title" — axis titles
8. "data" — stroked data lines, symbols, and error bars
9. "key" — graph legend (key)

Because drawing order is governed by these layers rather than the order in which internal `do*()` methods are executed, grid lines always appear behind data and axes labels always remain legible on top of grid lines. If you draw custom annotations onto a graph canvas `g`, you can target any layer directly via `g.layer("filldata").fill(...)` or `g.layer("data").stroke(...)`.

9.3 Module `graph.graph`: Graph Geometry

The classes `graphxy`, `graphx`, and `graphxyz` are defined in `graph.graph` and re-exported at the top level of `graph` as `graph.graphxy`, `graph.graphx`, and `graph.graphxyz`.

9.3.1 2D Graphs (`graphxy`)

```
class graph.graph.graphxy(xpos=0, ypos=0, width=None, height=None, ratio=goldenmean, key=None,
                           backgroundattrs=None, axesdist=0.8 * unit.v_cm, flipped=False, xaxisat=None,
                           yaxisat=None, **axes)
```

Constructs a two-dimensional x - y graph.

- `xpos`, `ypos`: Lower-left corner of the plot area on the graph canvas (in user units or PyX lengths; default 0).
- `width`, `height`, `ratio`: Dimensions of the rectangular plot area (excluding the space occupied by ticks, labels, and titles outside the box). At least `width` or `height` must be specified; if only one is given, the other is computed using `ratio` (which defaults to the golden ratio $(1 + \sqrt{5})/2 \approx 1.618$).
- `key`: Optional `graph.key.key` instance to generate a graph legend, or `None` (default) to omit the legend.
- `backgroundattrs`: List of fill/stroke attributes (e.g. `[deco.filled([color.gray(0.95)])]`) used to draw the rectangular plot background in the "background" layer, or `None` (default) for a transparent background.
- `axesdist`: Distance between multiple parallel axes stacked on the same side of the graph (default `0.8 * unit.v_cm`).
- `flipped`: When `True`, swaps the geometric orientation of the x and y axes so that the x -axis runs vertically and the y -axis runs horizontally (default `False`).
- `xaxisat`, `yaxisat`: Data value along the orthogonal axis (y or x , respectively) at which the main x or y axis should be positioned inside the plot (for example, `xaxisat=0`, `yaxisat=0` to cross the axes at the origin). Setting `xaxisat` or `yaxisat` disables automatic creation of the opposite linked axis (x_2 or y_2).

- ****axes**: Axis instances or underscore-split axis parameters. Valid axis names are `x`, `x2`, `x3`, ... and `y`, `y2`, `y3`, ...
 - If neither `x` nor `x2` is specified, `x` defaults to `graph.axis.linear`.
 - If `x2` (or `y2`) is not specified and `xaxisat` (or `yaxisat`) is `None`, PyX automatically creates a `graph.axis.linkedaxis` at the opposite side of the plot box (top for `x2`, right for `y2`). Pass `x2=None` or `y2=None` to suppress the top or right axis.
 - Odd-numbered axes (`x`, `x3`, ... and `y`, `y3`, ...) are placed at the bottom and left of the plot; even-numbered axes (`x2`, `x4`, ... and `y2`, `y4`, ...) are placed at the top and right, spaced outward by `axesdist`.
 - Any keyword argument of the form `<axisname>_<option>=...` (such as `x_min=0` or `yPainter_gridattr=...`) is split via `utils.kwsplit()` and applied to the corresponding axis.

axes

Dictionary mapping axis names ("`x`", "`y`", "`x2`", "`y2`", ...) to their `graph.axis.anchoredaxis` instances.

9.3.2 Plotting Data and Accessing Plot Items

`graphxy.plot(data, styles=None, rangewarning=1)`

Adds *data* (a `graph.data` instance, or an iterable of data instances) to the graph to be plotted with *styles* (a list of `graph.style` instances, or `None` to use `data.defaultstyles`).

Returns a `plotitem` instance (or a list of `plotitem` instances if *data* was an iterable). When the exact same *styles* list object is reused across multiple `plot()` calls on the same graph, the styles automatically cycle through their changeable attributes (such as symbol shapes or dash patterns).

Tip

Accessing plotted paths: After calling `g.doplot()` (or `g.finish()`), a `plotitem` provides attribute access to data stored by its styles. In particular, when plotting with `graph.style.line`, `p = g.plot(...)`; `g.doplot()`; `p.path` returns the plotted `path.path` in canvas coordinates—ideal for filling the area between two curves using `path.path.joined()`!

9.3.3 Graph Lifecycle and Finalization Methods

A graph accumulates data across `plot()` calls and automatically finalizes its layout and drawing when written to a file or when its `bbox()` is queried. If you want to query data positions (`pos()`) or axis paths before writing the file, call `dolayout()` (or `finish()`) explicitly. Calling any `do*` method multiple times is safe (subsequent calls are no-ops):

`graphxy.doranges()`

Computes automatic axis ranges from all plotted data sources and evaluates dynamic data sources (such as `graph.data.function`).

`graphxy.dolayout()`

Runs `doranges()`, partitions and paints all axes, and fixes the geometry of the graph. After `dolayout()` has run, coordinate conversion methods such as `pos()` are ready to use.

`graphxy.dobackground()`

Draws the graph background if `backgroundattr` was specified.

`graphxy.doaxes()`

Inserts the painted axis layers into the graph canvas.

`graphxy.dostyles()`

Selects changeable style attributes across all plotted datasets. Once `dostyles()` has been called, no further `plot()` calls are allowed.

`graphxy.doplotitem(plotitem)`

Draws a single *plotitem* onto the graph canvas.

`graphxy.doplot()`

Draws all remaining plot items onto the graph canvas.

`graphxy.dokeyitem(plotitem)`

Registers *plotitem* with the graph legend.

`graphxy.dokey()`

Paints and inserts the graph legend (*graph.key.key*).

`graphxy.finish()`

Executes all pending lifecycle steps (`dobackground()`, `doaxes()`, `doplot()`, and `dokey()`). Called automatically before output or bounding-box calculation.

9.3.4 Coordinate Queries and Axis Positioning

`graphxy.pos(x, y, xaxis=None, yaxis=None)`

`graphxy.pos_pt(x, y, xaxis=None, yaxis=None)`

Converts data values (*x*, *y*) along *xaxis* and *yaxis* (defaulting to `self.axes["x"]` and `self.axes["y"]`) into canvas coordinates (*xpos*, *ypos*) (in user units for `pos()`, or PostScript points for `pos_pt()`). Requires `dolayout()` to have been called if axis ranges are automatic.

`graphxy.vpos(vx, vy)`

`graphxy.vpos_pt(vx, vy)`

Converts normalized graph coordinates (*vx*, *vy*) in $[0, 1]^2$ into canvas coordinates (*xpos*, *ypos*).

`graphxy.vgeodesic(vx1, vy1, vx2, vy2)`

`graphxy.vgeodesic_el(vx1, vy1, vx2, vy2)`

Returns the geodesic path (`vgeodesic()`) or path element (`vgeodesic_el()`) connecting (*vx1*, *vy1*) to (*vx2*, *vy2*) in normalized graph coordinates (a straight line segment for *graphxy*).

`graphxy.axisatv(axis, v)`

Translates the anchored *axis* to the normalized graph coordinate *v* along the orthogonal direction (must be called after `dolayout()`).

In addition, for the primary *x* and *y* axes, *graphxy* exposes shortcut methods `Xbasepath`, `Xvbasepath`, `Xgridpath`, `Xvgridpath`, `Xtickpoint`, `Xtickpoint_pt`, `Xvtickpoint`, `Xvtickpoint_pt`, `Xtickdirection`, and `Xvtickdirection` (where *X* is *x* or *y*) that delegate directly to `self.axes["x"]` and `self.axes["y"]` (see `graph.axis.anchoredaxis`).

9.3.5 1D Graphs and Color Bars (*graphx*)

```
class graph.graph.graphx(xpos=0, ypos=0, length=None, size=0.5 * unit.v_cm, direction='vertical',
                          key=None, backgroundattrs=None, axesdist=0.8 * unit.v_cm, **kwargs)
```

A one-dimensional specialization of [graphxy](#) with a single active axis dimension (`x`, `x2`, ...) of length `length` and cross-sectional width `size` (default `0.5 * unit.v_cm`), oriented either "vertical" (default) or "horizontal". [graphx](#) is used automatically as the color scale legend (keygraph) for [graph.style.rect](#), [graph.style.surface](#), and [graph.style.density](#) plots.

9.3.6 3D Graphs ([graphxyz](#))

```
class graph.graph.graphxyz(xpos=0, ypos=0, size=None, xscale=1, yscale=1, zscale=1 / goldenmean,
                             xy12axesat=None, xy12axesatname='z', projector=central(10, -30, 30),
                             axesdist=0.8 * unit.v_cm, key=None, **axes)
```

Constructs a three-dimensional x - y - z graph projected onto the 2D canvas.

- `xpos`, `ypos`: Origin of the projected graph on the canvas.
- `size`: Overall scaling length of the projected 3D box.
- `xscale`, `yscale`, `zscale`: Relative scale factors along the three dimensions (default 1, 1, and 1 / goldenmean).
- `xy12axesat`, `xy12axesatname`: Moves the bottom x - y plane (containing axes `x`, `x2`, `y`, `y2`) to the specified data value along axis `xy12axesatname` (default "z").
- `projector`: A 3D-to-2D projection instance—either [central](#) (perspective projection) or [parallel](#) (orthographic/parallel projection), also accessible as class attributes `graphxyz.central` and `graphxyz.parallel`.
- `axesdist`: Distance between stacked axes in the same position.
- `**axes`: Supports up to four axes per dimension (`x`, `x2`, `x3`, `x4`, `y`, `y2`, `y3`, `y4`, `z`, `z2`, `z3`, `z4`). `x`, `x2`, `y`, `y2` border the bottom x - y plane (with `x` and `y` automatically placed along the front edges closest to the viewer); `x3`, `x4`, `y3`, `y4` border the top x - y plane; and `z`, `z2`, `z3`, `z4` run along the four vertical corners.

In addition to 3D versions of `pos()`, `vpos()`, `vgeodesic()`, and `vgeodesic_el()` (each taking three coordinates x , y , z), [graphxyz](#) provides depth and surface-normal angle queries used for occlusion sorting and 3D lighting:

vzindex(`vx`, `vy`, `vz`)

Returns the normalized depth of (`vx`, `vy`, `vz`) in $[-1, 1]$, where larger values are closer to the viewer.

vangle(`vx1`, `vy1`, `vz1`, `vx2`, `vy2`, `vz2`, `vx3`, `vy3`, `vz3`)

Returns the cosine of the angle between the viewing ray through (`vx1`, `vy1`, `vz1`) and the plane spanned by the three points (`vx1`, `vy1`, `vz1`), (`vx2`, `vy2`, `vz2`), and (`vx3`, `vy3`, `vz3`).

```
class central(distance, phi, theta, anglefactor=math.pi / 180)
```

Central (perspective) 3D-to-2D projector.

- `distance`: Distance of the viewer from the center of the graph in units of `xscale` / `yscale` / `zscale` (values around 10 give natural perspective; larger values approach parallel projection).
- `phi`: Azimuthal viewing angle in the x - y plane (in degrees by default).
- `theta`: Elevation angle above the x - y plane (in degrees by default).
- `anglefactor`: Conversion factor from the angle unit of `phi` and `theta` to radians (default `math.pi / 180`).

```
class parallel(phi, theta, anglefactor=math.pi / 180)
```

Parallel (orthographic) 3D-to-2D projector with azimuthal angle `phi` and elevation angle `theta`.

9.4 Module *graph.data*: Graph Data Sources

The classes in *graph.data* prepare tabular or functional data for `graph.graphxy.plot()`.

9.4.1 Tabular Data from Python Lists and Dictionaries

class `graph.data.values`(*title*='user provided values', ***columns*)

Creates a dataset from keyword arguments where each keyword name defines a column name and its value is a sequence (list, tuple, or NumPy array) of column values:

```
g.plot(graph.data.values(x=[1, 2, 3, 4], y=[1, 4, 9, 16], title="Squares"))
```

class `graph.data.points`(*points*, *title*='user provided points', *addlinenumbers*=1, ***columns*)

Creates a dataset from a list of rows (*points*), where each row is a sequence of values [*v1*, *v2*, ...].

- ***columns*: Maps column names (such as *x*=1, *y*=2) to column indices in each row.
- *addlinenumbers*: When true (default), 1-based row numbers are placed in virtual column 0, and data columns in *points* are indexed starting at 1. When false, data columns in *points* are indexed starting at 0.

9.4.2 Reading Data from Files

class `graph.data.file`(*filename*, *commentpattern*=defaultcommentpattern,
 columnpattern=defaultcolumnpattern, *stringpattern*=defaultstringpattern, *skiphead*=0,
 skiptail=0, *every*=1, *title*=notitle, *context*={}, *copy*=1, *replacedollar*=1,
 columncallback='__column__', ***columns*)

Reads whitespace-separated or custom-formatted tabular data from *filename* (a file path or open text file object). Parsed files are cached in memory so multiple *file* instances reading different columns from the same file only parse the file once.

Line and column parsing:

- Lines matching *commentpattern* (by default, lines starting with #, !, or %, as well as blank lines) are skipped as comments.
- If the last non-empty comment line immediately preceding the first data line contains tokens, those tokens are used as *file column names*.
- Each data line is tokenized by repeatedly matching *stringpattern* (which by default extracts double-quoted strings "... " containing spaces) and *columnpattern* (which extracts whitespace-delimited tokens and converts them to float when possible, or keeps them as strings otherwise).
- *skiphead* and *skiptail* skip the specified number of data rows at the beginning or end of the file, and *every* keeps only every *every*-th row.

Column definitions and mathematical expressions (***columns*):

Each keyword argument *name*=*spec* defines a named column passed to the plotting styles:

- If *spec* is an integer, it selects the 1-based column number from the file (or 0 for the 1-based data row number).
- If *spec* is a string, it can be either a file column name (from the header comment) or an arbitrary Python mathematical expression evaluated for each row. Inside the expression, you can reference file columns by their header names or via \$1, \$2, ... (when *replacedollar*=1), as well as all functions and constants in Python's *math* module (plus the built-in helpers listed in the table below) and any variables passed in the *context* dictionary.

- When `copy=1` (default), all named columns from the file header are also passed through to the plotting styles automatically unless overridden by `**columns`.
- `title`: Legend title for the dataset. By default, PyX constructs a title automatically from `filename` and the column expressions; pass `title=None` to suppress the legend entry.

Example:

Given `test.dat`:

```
# A   B C
1.234 1 2
5.678 3 4
```

calling:

```
d = graph.data.file("test.dat", a=1, b="B", c="2*B+$3")
```

produces columns `a = [1.234, 5.678]`, `b = [1.0, 3.0]`, and `c = [4.0, 10.0]` (along with A, B, and C since `copy=1`).

The default regular expression patterns on `file` are:

defaultcommentpattern

```
re.compile(r"(\#|!+|#+)\s*")
```

defaultstringpattern

```
re.compile(r"\"(.*)\"(\s+|$)")
```

defaultcolumnpattern

```
re.compile(r"(.*)\"(\s+|$)")
```

9.4.3 Transforming Existing Datasets and Other File Formats

```
class graph.data.data(data, title=notitle, context={}, copy=1, replacedollar=1,
                      columncallback='__column__', **columns)
```

Wraps an existing `graph.data` instance `data` and derives new columns or mathematical expressions from its columns using the same keyword syntax as `file`.

```
class graph.data.confirfile(filename, title=notitle, context={}, copy=1, replacedollar=1,
                           columncallback='__column__', **columns)
```

Reads tabular data from an INI-style configuration file parsed by Python's `configparser` module. Each `[section]` in the file becomes one data row, with `option = value` entries acting as named columns.

```
class graph.data.cbdfirfile(filename, minrank=None, maxrank=None, title=notitle, context={}, copy=1,
                           replacedollar=1, columncallback='__column__', **columns)
```

Reads vector coastline/boundary segments from a CIA World DataBank (`.cbd`) binary file, filtering segments by rank between `minrank` and `maxrank`.

9.4.4 Analytical and Parametric Functions

```
class graph.data.function(expression, title=notitle, min=None, max=None, points=100, context={})
```

Generates data points by evaluating a mathematical function of one variable.

- `expression`: String assignment specifying the dependent axis variable, the independent axis variable in parentheses, and the formula—for example, `"y(x) = sin(x) / x"` or `"x(y) = y**2"`.

- *min*, *max*: Domain of the independent variable. If omitted, the range of the corresponding graph axis is used automatically.
- *points*: Number of sample points (default 100), spaced uniformly in graph coordinates (so a logarithmic axis is sampled logarithmically).
- *title*: Legend title (defaults to *expression*; set to None to hide).
- *context*: Dictionary of external Python variables or callables accessible within *expression*.

class `graph.data.paramfunction`(*varname*, *min*, *max*, *expression*, *title=notitle*, *points=100*, *context={}*)

Generates data points from a parametric curve in 2D or 3D as the parameter *varname* varies linearly from *min* to *max* over *points* samples.

- *expression*: Tuple assignment mapping the parameter *varname* to two or three axis coordinates—for example, "x, y = cos(t), sin(t)" or "x, y, z = cos(t), sin(t), 0.1*t".

9.4.5 Built-in Functions in Mathematical Expressions

All mathematical expressions in *file*, *data*, *function*, and *paramfunction* have access to every function and constant in Python's standard math module (sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, tanh, exp, log, log10, sqrt, hypot, pi, e, etc.), plus the following convenience functions:

Name	Definition
<code>neg(x)</code>	$-x$
<code>abs(x)</code>	$\text{abs}(x)$
<code>sgn(x)</code>	-1 if $x < 0$ else 1
<code>sind(x)</code>	$\text{math.sin}(\text{math.pi} / 180 * x)$ (sine in degrees)
<code>cosd(x)</code>	$\text{math.cos}(\text{math.pi} / 180 * x)$ (cosine in degrees)
<code>tand(x)</code>	$\text{math.tan}(\text{math.pi} / 180 * x)$ (tangent in degrees)
<code>asind(x)</code>	$180 / \text{math.pi} * \text{math.asin}(x)$ (in degrees)
<code>acosd(x)</code>	$180 / \text{math.pi} * \text{math.acos}(x)$ (in degrees)
<code>atand(x)</code>	$180 / \text{math.pi} * \text{math.atan}(x)$ (in degrees)
<code>norm(x, y)</code>	$\text{math.hypot}(x, y)$
<code>splitatvalue</code>	See <i>splitatvalue()</i> below

graph.data.splitatvalue(*value*, **splitpoints*)

Splits a coordinate value into discontinuous sections at *splitpoints* (useful with split axes; see `graph.axis.split`). Returns a tuple (section, value). With a single split point *s1*, section is 0 when *value* $\leq$ *s1* and 1 otherwise. With multiple split points, section alternates between 0, None (ignored gap between the first and second split points), 1, None, 2, etc.

9.5 Module *graph.style*: Graph Styles

Graph styles control how data columns are rendered visually inside a graph. The *styles* argument of `graph.graphxy.plot()` accepts a list of style instances, allowing you to combine multiple styles for a single dataset—for example, [`graph.style.line()`, `graph.style.symbol()`, `graph.style.errorbar()`].

9.5.1 Hidden Data-Provider Styles

Several styles (*pos*, *range*, *barpos*, *stackedbarpos*, and *gridpos*) are *hidden styles*: they do not draw marks themselves (except for optional baseline paths), but instead read data columns and compute graph-coordinate positions in a shared *sharedata* object consumed by visible styles. Because PyX automatically inserts the required default hidden style whenever a visible style is used, you only need to instantiate a hidden style explicitly when you want to customize its column name mapping (*username*s) or baseline (*fromvalue*).

class `graph.style.pos`(*usenames*={}, *epsilon*=1e-10)

Hidden style that reads one data column per graph axis (matching the axis name "x", "y", "z", or mapped via the dictionary *usenames*) and computes the normalized graph position. Points outside [0, 1] by more than *epsilon* are marked as out of bounds.

class `graph.style.range`(*usenames*={}, *epsilon*=1e-10)

Hidden style that reads interval/uncertainty bounds for one or more axes X (or mapped via *usenames*). For each axis X, you can supply any of the following column combinations:

9.5.2 Symbols, Lines, Impulses, and Error Bars

class `graph.style.symbol`(*symbol*=*changecross*, *size*=0.2 * *unit.v_cm*, *symbolattrs*=[])

Draws a marker symbol at each data point. This is the default style for all tabular data sources (*file*, *values*, *points*, etc.).

- *symbol*: A symbol drawing callable with signature `symbol(c, x_pt, y_pt, size_pt, attrs)` or a changeable attribute cycling through symbol callables.
- *size*: Visual diameter of the symbol (default 0.2 * `unit.v_cm`).
- *symbolattrs*: Stroke/fill/color attributes merged with `defaultsymbolattrs` (`[deco.stroked]`).

Predefined symbol shapes (class attributes on *symbol*):

- `symbol.cross` — diagonal cross × (stroke only)
- `symbol.plus` — orthogonal plus + (stroke only)
- `symbol.square` — square (can be stroked, filled, or both)
- `symbol.triangle` — equilateral triangle (can be stroked, filled, or both)
- `symbol.circle` — circle (can be stroked, filled, or both)
- `symbol.diamond` — diamond Rhombus (can be stroked, filled, or both)

Changeable symbol lists (automatically cycle across datasets when the same *symbol* instance is reused):

- `symbol.changecross` — `attr.changelist([cross, plus, square, triangle, circle, diamond])`
- `symbol.changeplus` — `attr.changelist([plus, square, triangle, circle, diamond, cross])`
- `symbol.changesquare` — `attr.changelist([square, triangle, circle, diamond, cross, plus])`
- `symbol.changetriangle` — `attr.changelist([triangle, circle, diamond, cross, plus, square])`
- `symbol.changecircle` — `attr.changelist([circle, diamond, cross, plus, square, triangle])`
- `symbol.changediamond` — `attr.changelist([diamond, cross, plus, square, triangle, circle])`
- `symbol.changesquaretwice` — repeats each closed shape twice: `[square, square, triangle, triangle, circle, circle, diamond, diamond]`
- `symbol.changetriangletwice` — `[triangle, triangle, circle, circle, diamond, diamond, square, square]`
- `symbol.changecircletwice` — `[circle, circle, diamond, diamond, square, square, triangle, triangle]`

- `symbol.changediamondtwice` — `[diamond, diamond, square, square, triangle, triangle, circle, circle]`

Changeable stroke/fill decorators (useful in *symbolattrs* alongside the **twice* symbol lists above):

- `symbol.changestrokedfilled` — `attr.changelist([deco.stroked, deco.filled])`
- `symbol.change-filledstroked` — `attr.changelist([deco.filled, deco.stroked])`

class `graph.style.line`(*lineattrs=[]*, *epsilon=1e-10*)

Connects consecutive data points with geodesic segments clipped to the graph boundary. This is the default style for `graph.data.function` and `graph.data.paramfunction`.

- *lineattrs*: Stroke attributes merged with defaultlineattrs (`[line.changelinestyle]`).
- `line.changelinestyle`: `attr.changelist([style.linestyle.solid, style.linestyle.dashed, style.linestyle.dotted, style.linestyle.dashdotted])`.

class `graph.style.impulses`(*lineattrs=[]*, *fromvalue=0*, *frompathattrs=[]*, *valueaxisindex=1*)

Draws vertical (when *valueaxisindex=1*) or horizontal (when *valueaxisindex=0*) impulse stems from a baseline value *fromvalue* (or from the bottom/left graph boundary if *fromvalue=None*) to each data point. When *fromvalue* is not *None*, the baseline is stroked using *frompathattrs*.

class `graph.style.errorbar`(*size=0.1 * unit.v_cm*, *errorbarattrs=[]*, *epsilon=1e-10*)

Draws error bars with perpendicular end caps of visual length *size* using the interval bounds provided by `range` (such as *dy*, *ymin/ymax*, or *dymn/dymax*). Error bars extending outside the plot area are clipped to the graph boundary and their out-of-bounds caps are omitted.

9.5.3 Text Labels and Vector Arrows

class `graph.style.text`(*textname='text'*, *dxname=None*, *dyname=None*, *dxunit=0.3 * unit.v_cm*, *dyunit=0.3 * unit.v_cm*, *textdx=0 * unit.v_cm*, *textdy=0.3 * unit.v_cm*, *textattrs=[]*)

Places a text label (read from column *textname*) at each data point, offset by (*textdx*, *textdy*)—or by per-point offsets read from columns *dxname* and *dyname* multiplied by *dxunit* and *dyunit*. *textattrs* is merged with `[text.halign.center, text.vshift.mathaxis]`.

class `graph.style.arrow`(*linelength=0.25 * unit.v_cm*, *arrowsize=0.15 * unit.v_cm*, *lineattrs=[]*, *arrowattrs=[]*, *arrowpos=0.5*, *epsilon=1e-10*, *decorator=deco.earrow*)

Draws 2D vector field arrows at each data point using two additional data columns:

- *size*: Scale factor multiplying both *linelength* and *arrowsize*.
- *angle*: Direction angle in degrees counter-clockwise from the positive *x*-axis.

arrowpos controls how each arrow segment is anchored at the data point (0 starts at the point, 0.5 centers the arrow on the point, and 1 ends at the point).

9.5.4 Histograms and Bar Charts

class `graph.style.histogram`(*lineattrs=[]*, *steps=0*, *fromvalue=0*, *frompathattrs=[]*, *fillable=0*, *rectkey=0*, *autohistogramaxisindex=0*, *autohistogrampointpos=0.5*, *epsilon=1e-10*)

Plots a 2D histogram.

- *lineattrs*: Attributes used to draw the histogram outline (merged with `[deco.stroked]`).
- *steps*: When 0 (default), draws full vertical box outlines for each bin; when 1, draws only the upper step contour.
- *fromvalue*, *frompathattrs*: Baseline value and stroke attributes for the histogram base (or *fromvalue=None* to start at the axis boundary).

- *fillable*: When true, closes the histogram path along the baseline and graph boundaries so that passing `deco.filled([...])` in *lineattrs* cleanly fills the area under the histogram.
- *rectkey*: When true, draws a filled/stroked rectangle in the graph legend instead of a line segment.

Specifying bin widths (equal vs. variable bin widths):

- *Equal-width bins (automatic)*: If you supply only point coordinates *x* and *y* (with at least two data points), *histogram* automatically computes equal bin widths along axis *autohistogramaxisindex* (0 for *x*, 1 for *y*). *autohistogrampointpos* specifies where the given data coordinate sits within each bin: 0.5 (default) centers each bin on the data point, 0 places the bin to the right of the data point, and 1 places the bin to the left.
- *Variable-width bins (explicit ranges)*: To plot a histogram with unequal bin widths, supply range columns for the bin axis (such as *xmin* and *xmax*, or *x* and *dx*, as consumed by *range*) alongside the height column *y*. Whenever range columns are present for the bin axis, *histogram* uses the exact interval $[x_{\min}, x_{\max}]$ of each bin.

class `graph.style.barpos`(*fromvalue=None, frompathattrs=[], epsilon=1e-10*)

Hidden style that computes bar positions along a `graph.axis.bar` (or `graph.axis.nestedbar`) axis. For a bar axis named *X* (e.g. *x*), the categorical bar names are read from column *Xname* (e.g. *xname*) and the bar heights are read from the continuous axis column (e.g. *y*). To plot grouped bars side by side on a `graph.axis.nestedbar` axis, pass a 2-tuple (*category*, *subcategory*) in *Xname* (for example, *xname*="\$1, 0" and *xname*="\$1, 1").

class `graph.style.stackedbarpos`(*stackname, addontop=0, epsilon=1e-10*)

Hidden style that stacks a new bar series on top of the previous bar series, reading the upper coordinate (or the incremental height when *addontop*=1) from column *stackname*.

class `graph.style.bar`(*barattrs=[], epsilon=1e-10, gradient=color.gradient.RedBlack*)

Draws filled rectangular (2D) or prism (3D) bars using the positions computed by *barpos* or *stackedbarpos*. *barattrs* is merged with `[color.gradient.Rainbow, deco.stroked([color.gray.black])]`, so multiple bar series in a grouped or stacked bar chart automatically receive distinct colors from `color.gradient.Rainbow`.

class `graph.style.changebar`(*barattrs=[], epsilon=1e-10, gradient=color.gradient.RedBlack*)

Variant of *bar* that cycles the changeable attributes in *barattrs* from bar to bar *within* a single dataset rather than across multiple datasets.

9.5.5 Rectangles, Grids, Surfaces, and Density Plots

class `graph.style.rect`(*colorname='color', gradient=color.gradient.Grey, coloraxis=None, keygraph=_autokeygraph*)

Fills axis-aligned rectangles (defined by *range* bounds on *x* and *y*) with colors mapped from column *colorname* via *gradient*.

- *coloraxis*: Axis instance (defaults to `graph.axis.linear`) mapping values in *colorname* to $[0, 1]$.
- *keygraph*: A `graph.graphx` instance drawn beside the main graph as a color bar legend. By default, a vertical `graph.graphx` is created automatically; pass *keygraph*=None to omit the color bar.

class `graph.style.gridpos`(*index1=0, index2=1, epsilon=1e-10*)

Hidden style that organizes point data along graph dimensions *index1* and *index2* into a structured 2D rectangular mesh.

class `graph.style.grid`(*gridlines1=1, gridlines2=1, gridattrs=[], **kwargs*)

Strokes a wireframe mesh in 2D or 3D along the first (*gridlines1*) and/or second (*gridlines2*) mesh directions provided by *gridpos*.

```
class graph.style.surface(gridlines1=0.05, gridlines2=0.05, gridcolor=None, backcolor=color.gray.black,
                        **kwargs)
```

Draws a smoothly shaded 2D or 3D surface over the rectangular mesh provided by [gridpos](#) using a Gouraud-shaded triangle mesh (each grid cell is subdivided into four triangles).

- *gridlines1*, *gridlines2*, *gridcolor*: If *gridcolor* is set, draws narrow triangle strips of fractional width *gridlines1* and *gridlines2* in *gridcolor* along the cell boundaries.
- *backcolor*: Color used for triangles facing away from the viewer (or `None` to color both sides identically).
- **Coloring**: Accepts the same *colorname*, *gradient*, *coloraxis*, and *keygraph* arguments as [rect](#). If no *colorname* column is present in the data, [surface](#) automatically computes 3D lighting from the viewing angle and distance using `lighting()`. Note that Gouraud shading meshes in PostScript/PDF require Gray, RGB, or CMYK colors; wrap HSB gradients in [color.rgbgradient](#) or [color.cmykgradient](#).

```
class graph.style.density(epsilon=1e-10, **kwargs)
```

Renders a 2D pseudocolor heatmap as an embedded [bitmap](#) image over an equidistantly spaced grid (in graph coordinates). Accepts the same *colorname*, *gradient*, *coloraxis*, and *keygraph* arguments as [rect](#). Because [density](#) embeds a compact raster bitmap rather than a triangle mesh, it produces much smaller files than [surface](#) for large 2D grids and avoids PDF viewer anti-aliasing seams.

9.6 Module [graph.key](#): Graph Keys (Legends)

```
class graph.key.key(dist=0.2 * unit.v_cm, pos='tr', hpos=None, vpos=None, hinside=1, vinside=1, hdist=0.6 *
                  unit.v_cm, vdist=0.4 * unit.v_cm, symbolwidth=0.5 * unit.v_cm, symbolheight=0.25 *
                  unit.v_cm, symbolspace=0.2 * unit.v_cm, textattrs=[], columns=1, columndist=0.5 *
                  unit.v_cm, border=0.3 * unit.v_cm, keyattrs=None)
```

Creates a graph legend to be passed to the *key* argument of [graph.graphxy](#) or [graph.graphxyz](#).

- *pos*: Two-character alignment string positioning the legend relative to the plot box: vertical position "t" (top), "m" (middle), or "b" (bottom) followed by horizontal position "l" (left), "c" (center), or "r" (right). Default is "tr" (top right).
- *hpos*, *vpos*: Explicit relative horizontal and vertical positions in $[0, 1]$ (alternative to *pos*; cannot be combined with *pos*).
- *hinside*, *vinside*: 1 (default) to place the legend inside the plot box, or 0 to place it outside the plot box (for example, *pos*="tl", *hinside*=0 places the legend to the left of the top-left corner).
- *hdist*, *vdist*: Horizontal and vertical margin between the legend and the reference corner of the plot box.
- *dist*: Vertical baseline separation between consecutive entries.
- *symbolwidth*, *symbolheight*, *symbolspace*: Dimensions of the style preview swatch and horizontal gap between the swatch and the dataset title.
- *textattrs*: Text attributes applied to legend labels (merged with `[text.vshift.mathaxis]`).
- *columns*, *columndist*: Number of legend columns (default 1) and horizontal spacing between columns.
- *keyattrs*, *border*: Optional draw attributes (such as `[deco.filled([color.gray.white])]`, `deco.stroked()`) used to draw a box around the legend, padded on all sides by *border* (default $0.3 * \text{unit.v_cm}$).

10.1 Component Architecture

Axes in PyX convert data values into normalized graph coordinates $v \in [0, 1]$ and render the visual axis elements (baseline, tick marks, grid lines, tick labels, and axis title). Although most commonly used inside a `graph.graphxy` or `graph.graphxyz`, axes can also be drawn standalone along any arbitrary path using `graph.axis.pathaxis()`.

Internally, each axis delegates its work to six modular components located in submodules of `graph.axis`:

axis (`graph.axis.axis`)

Defines the mathematical mapping between data values and normalized graph coordinates $[0, 1]$, orchestrates automatic range adjustment, and combines the partitioner, texter, painter, and rater to choose the optimal tick layout.

Because an axis instance itself is stateless, a graph wraps each axis in an `anchoredaxis` container that pairs the axis with a geometric positioner and holds per-graph range and tick data.

tick (`graph.axis.tick`)

Represents individual ticks and subticks placed along an axis, using exact rational arithmetic (`rational`) to avoid floating-point rounding artifacts.

parter (`graph.axis.parter`)

Short for *partitioner*: generates one or more candidate lists of tick and subtick positions for a given data range.

texter (`graph.axis.texter`)

Formats numeric or rational tick positions into label strings (using decimal, scientific/exponential, or fraction notation).

painter (`graph.axis.painter`)

Paints the axis baseline, ticks, grid lines, tick labels, and axis title onto an axis canvas.

rater (`graph.axis.rater`)

Evaluates candidate tick partitionings by scoring the number of ticks/labels and checking the painted label boxes for overlaps or crowding. The candidate with the lowest penalty score is selected.

positioner (`graph.axis.positioner`)

Defines the geometric curve (straight line or arbitrary path), tick direction, and grid paths of an anchored axis on the canvas.

10.1.1 Customizing Axes and Subcomponents via Keyword Splitting

Every axis, painter, and texter instance in `graph.axis` is callable (implementing `__call__` via `utils.merge_members_kwargs()`) and supports underscore-separated keyword forwarding (`utils.kwsplit()`). This means you can customize nested components without verbose boilerplate:

```

from pyx import *

# Equivalent ways to enable grid lines and set a custom label prefix on a linear axis:
ax1 = graph.axis.linear(min=0, max=10,
                        painter=graph.axis.painter.regular(gridattrs=[color.gray(0.8)]),
                        texter=graph.axis.texter.default(mantissatexter_prefix=r"\$"))

ax2 = graph.axis.linear(min=0, max=10,
                        painter_gridattrs=[color.gray(0.8)],
                        texter_mantissatexter_prefix=r"\$")

```

10.2 Module *graph.axis.axis*: Axis Classes

All classes in *graph.axis.axis* are also exposed directly in *graph.axis* (e.g. *graph.axis.linear*, *graph.axis.log*, *graph.axis.bar*).

10.2.1 Continuous Regular Axes (*linear* and *logarithmic*)

```

class graph.axis.axis.linear(min=None, max=None, reverse=0, divisor=None, title=None,
                             parter=parter.autolinear(), manualticks=[], rater=rater.linear(), maxworse=2,
                             texter=texter.default(), painter=painter.regular(), linkpainter=painter.linked(),
                             fallbackrange=None, **kwargs)

```

A linear axis mapping $[min, max]$ linearly onto $[0, 1]$. Also available under the short alias *lin*.

- *min*, *max*: Fixed lower and upper bounds of the axis. When left as *None*, the bounds are determined automatically from the plotted data and extended to the nearest outer tick if allowed by *parter*. If *min* > *max*, the bounds are swapped and *reverse* is toggled automatically.
- *reverse*: When true, reverses the direction of the axis so values decrease from 0 to 1 in graph coordinates.
- *divisor*: Optional number by which all data values are divided before partitioning and formatting ticks (for example, *divisor=math.pi* paired with *texter=graph.axis.texter.rational(suffix=r"\pi")* labels ticks in multiples of π).
- *title*: Axis title string, typeset using the graph's text engine.
- *parter*: Partitioner instance (default *parter.autolinear*) or *None* to draw only *manualticks*.
- *manualticks*: List of *tick* instances merged with the ticks generated by *parter* (manual ticks override automatic ticks at identical positions).
- *rater*, *maxworse*: Rater instance (*rater.linear*) and maximum number of consecutively worse partitioning candidates to explore before stopping the search in each direction.
- *texter*: Texter instance (default *texter.default*) used to format unlabeled ticks.
- *painter*: Painter instance (default *painter.regular*) used to draw the primary axis.
- *linkpainter*: Painter instance (default *painter.linked*) used when an opposite linked axis (such as *x2* or *y2*) is created automatically.
- *fallbackrange*: Minimal range width (or (*min*, *max*) tuple) used when the plotted data has zero span (for example, a constant function).
- ***kwargs*: Underscore-split keyword arguments forwarded to *painter*, *texter*, *linkpainter*, *parter*, or *rater* (e.g. *painter_gridattrs=...*).

```
class graph.axis.axis.lin(...)
```

Alias for [linear](#).

```
class graph.axis.axis.logarithmic(min=None, max=None, reverse=0, divisor=None, title=None,
                                  parter=parter.autologarithmic(),
                                  linearparter=parter.autolinear(extendtick=None), manualticks=[],
                                  maxworse=2, rater=rater.logarithmic(), texter=texter.default(),
                                  painter=painter.regular(), linkpainter=painter.linked(),
                                  fallbackrange=None, **kwargs)
```

A logarithmic axis mapping positive (or negative) values logarithmically onto $[0, 1]$. Also available under the short alias [log](#).

All arguments work as in [linear](#), with defaults tailored for logarithmic scales (`parter.autologarithmic` and `rater.logarithmic`). If the data spans such a narrow interval that no valid logarithmic partitioning without overlapping labels can be found, the axis logs a warning and automatically falls back to `linearparter` (unless `linearparter=None`).

```
class graph.axis.axis.log(...)
```

Alias for [logarithmic](#).

10.2.2 Discrete and Segmented Axes ([bar](#), [nestedbar](#), and [split](#))

```
class graph.axis.axis.bar(subaxes=None, defaultsubaxis=linear(painter=None, linkpainter=None,
                                                             parter=None), dist=0.5, firstdist=None, lastdist=None, title=None, reverse=0,
                          epsilon=1e-10, painter=painter.bar(), linkpainter=painter.linkedbar(), **kwargs)
```

Discrete categorical axis for bar charts ([graph.style.bar](#)). A bar axis maps a 2-tuple (`name`, `subvalue`) to a normalized graph coordinate:

- `name` selects one of the discrete categories along the bar axis (ordered by their appearance in the data, or reversed when `reverse=1`).
- `subvalue` is converted by the subaxis associated with `name` and mapped into the sub-interval of $[0, 1]$ allocated to that category.
- `subaxes`: Optional dictionary or sequence mapping category names to explicit subaxis instances; any category not in `subaxes` uses a copy of `defaultsubaxis`.
- `dist`, `firstdist`, `lastdist`: Spacing between consecutive category intervals, before the first category, and after the last category, measured relative to a unit subaxis size of 1 (default `dist=0.5`, with `firstdist` and `lastdist` defaulting to `0.5 * dist`).
- `**kwargs`: Underscore-split keyword arguments forwarded to `defaultsubaxis`, `painter`, or `linkpainter`.

```
class graph.axis.axis.nestedbar(subaxes=None, defaultsubaxis=bar(dist=0, painter=None,
                                                                linkpainter=None), dist=0.5, firstdist=None, lastdist=None, title=None,
                                reverse=0, painter=painter.bar(), linkpainter=painter.linkedbar(),
                                **kwargs)
```

A two-level hierarchical bar axis for grouped side-by-side bar charts, where `defaultsubaxis` is itself a [bar](#) axis with `dist=0`. Data values passed to a [nestedbar](#) axis have the nested tuple form `((category, subcategory), subvalue)`.

```
class graph.axis.axis.split(subaxes=None, defaultsubaxis=linear(), dist=0.5, firstdist=0, lastdist=0,
                           title=None, reverse=0, painter=painter.split(), linkpainter=painter.linkedsplit(),
                           **kwargs)
```

A segmented continuous axis divided into multiple continuous sub-ranges separated by visual break markers (painted by `painter.split`). Data values can be routed to subaxes automatically using [graph.data.splitatvalue\(\)](#).

```
class graph.axis.axis.sizedlinear(size=1, **kwargs)
```

```
class graph.axis.axis.autosizedlinear(**kwargs)
```

Subclasses of [linear](#) (also aliased as [sizedlin](#) and [autosizedlin](#)) for use as subaxes in a [bar](#) or [split](#) axis. [sizedlinear](#) assigns a fixed relative width *size* to the subaxis, whereas [autosizedlinear](#) scales the width of each subaxis proportionally to its data span $\text{max} - \text{min}$.

10.2.3 Anchored, Linked, and Standalone Path Axes

```
class graph.axis.axis.anchoredaxis(axis, textengine, errorname)
```

Combines an [_axis](#) instance with a [positioner.positioner](#) and per-graph storage ([self.data](#) and [self.canvas](#)). Graph instances store their anchored axes in [g.axes\[name\]](#). Once the axis layout has been created, [anchoredaxis](#) provides the following geometric methods:

```
basepath(x1=None, x2=None)
```

```
vbsepath(v1=None, v2=None)
```

Returns a [path.path](#) along the axis baseline between data values *x1* and *x2* (or graph coordinates *v1* and *v2*).

```
gridpath(x)
```

```
vgridpath(v)
```

Returns the grid line [path.path](#) across the graph at data value *x* (or graph coordinate *v*), or [None](#) if no grid path is defined.

```
tickpoint(x)
```

```
tickpoint_pt(x)
```

```
vtickpoint(v)
```

```
vtickpoint_pt(v)
```

Returns the canvas coordinates (*x*, *y*) of the point on the axis baseline corresponding to data value *x* (or graph coordinate *v*).

```
tickdirection(x)
```

```
vtickdirection(v)
```

Returns the unit normal vector (*dx*, *dy*) pointing toward the interior of the graph at data value *x* (or graph coordinate *v*).

```
class graph.axis.axis.linkedaxis(linkedaxis=None, errorname='manual-linked', painter=_marker,
                                  **kwargs)
```

An anchored axis that shares the data range, scaling, and tick positions of another [anchoredaxis](#) instance *linkedaxis* (for example, to link an axis in one graph to the axis of an adjacent graph in a multi-panel figure).

```
setlinkedaxis(linkedaxis)
```

Binds or updates the target *linkedaxis* after construction.

```
class graph.axis.axis.anchoredpathaxis(path, axis, textengine=text.defaulttextengine, direction=1)
```

```
graph.axis.axis.pathaxis(path, axis, **kwargs)
```

Creates and paints a standalone *axis* (which must have explicit *min* and *max* bounds) along an arbitrary geometric [path.path](#). *direction* specifies whether ticks point to the left (1, default) or right (-1) of the path tangent. [pathaxis\(\)](#) immediately builds the axis and returns its painted [canvas.canvas](#) so it can be inserted directly into any canvas:

```
c = canvas.canvas()
c.insert(graph.axis.pathaxis(path.curve(0, 0, 2, 2, 4, -2, 6, 0),
graph.axis.linear(min=0, max=10)))
```

10.3 Module *graph.axis.timeaxis*: Date and Time Axes

The *graph.axis.timeaxis* module provides classes for plotting Python `datetime.datetime` (or `datetime.date`) objects along a continuous axis:

class *graph.axis.timeaxis.timeaxis*(*partier=None, rater=rater.linear(), **args*)

A continuous axis (inheriting from *linear*) whose data bounds and values are Python `datetime.datetime` instances.

class *graph.axis.timeaxis.timetick*(*year, month, day, ticklevel=0, labellevel=0, label=None, labelattrs=[], **kwargs*)

A tick subclassing `datetime.datetime` with tick and label hierarchy levels *ticklevel* and *labellevel* and optional explicit *label*.

class *graph.axis.timeaxis.timetexter*(*format='%c'*)

Formats *timetick* positions using `datetime.datetime.strftime()` with the given *format* string.

10.4 Module *graph.axis.tick*: Axis Ticks

class *graph.axis.tick.rational*(*x, power=1, floatprecision=10*)

Exact rational number *num* / *denom* used internally by partitioners and texters to avoid floating-point rounding errors. *x* can be:

- A float (converted using precision *floatprecision*),
- A decimal or fraction string (such as "0.25" or "1/3"),
- A 2-tuple of integers (*num*, *denom*), or
- Any object with *.num* and *.denom* integer attributes (or a Python `fractions.Fraction`).

If *power* is given, the rational number is raised to the integer exponent *power*.

class *graph.axis.tick.tick*(*x, ticklevel=0, labellevel=0, label=None, labelattrs=[], power=1, floatprecision=10*)

Represents a tick mark and/or label at position *x* (interpreted as a *rational*). Instances of *tick* can be passed to the *manualticks* list of any regular axis:

```
ax = graph.axis.linear(min=0, max=4,
manualticks=[graph.axis.tick.tick(math.pi, label=r"$\pi$")])
```

- *ticklevel*: Non-negative integer specifying the tick hierarchy level (0 for a major tick, 1 for a subtick, 2 for a sub-subtick, etc.), or *None* to draw no tick mark at this position.
- *labellevel*: Non-negative integer specifying the label hierarchy level (0 for a major label, 1 for a sublabel, etc.), or *None* to omit the label at this position.
- *label*: Explicit label string, or *None* to let the axis's *texter* generate the label automatically.
- *labelattrs*: Additional text attributes applied to this specific tick's label.

10.5 Module *graph.axis.parter*: Axis Partitioners

Partitioners generate candidate lists of *tick* instances for a given axis range.

10.5.1 Linear Partitioners

class *graph.axis.parter.linear*(*tickdists=None, labeldists=None, extendtick=0, extendlabel=None, epsilon=1e-10*)

class *graph.axis.parter.lin*(...)

Places ticks at fixed equidistant intervals specified in *tickdists* (a list of intervals [*major_step*, *subtick_step*, ...], where each step can be a number, string, or *rational*).

- *labeldists*: Optional list of step sizes for labeled ticks. When *None* (default), labels are placed at major ticks (level 0) only.
- *extendtick*, *extendlabel*: Tick or label level (default 0 for *extendtick*, *None* for *extendlabel*) to which an auto-ranged axis is expanded if the data exceeds the outermost tick by more than *epsilon*.

class *graph.axis.parter.autolinear*(*variants=defaultvariants, extendtick=0, epsilon=1e-10*)

class *graph.axis.parter.autolin*(...)

Automatically generates candidate *linear* partitionings by scaling the base step patterns in *variants* by powers of 10 to match the data span.

autolinear.defaultvariants

[[1, 1/2], [2, 1], [5/2, 5/4], [5, 5/2]] (represented as *rational* pairs of [*tick_step*, *subtick_step*]).

10.5.2 Logarithmic Partitioners

class *graph.axis.parter.preexp*(*pres, exp*)

Specifies a repeating sequence of logarithmic tick positions of the form $p_i \cdot \exp^n$ for all integers n and prefixes $p_i \in \text{pres}$.

class *graph.axis.parter.logarithmic*(*tickpreexps=None, labelpreexps=None, extendtick=0, extendlabel=None, epsilon=1e-10*)

class *graph.axis.parter.log*(...)

Places ticks on a logarithmic axis according to the list of *preexp* instances in *tickpreexps* (for ticks, subticks, ...) and *labelpreexps* (for labels, sublabels, ...).

Predefined *preexp* constants on *logarithmic*:

- *logarithmic.pre1exp* — powers of 10 (10^n)
- *logarithmic.pre125exp* — $1, 2, 5 \times 10^n$
- *logarithmic.pre1to9exp* — $1, 2, \dots, 9 \times 10^n$

class *graph.axis.parter.autologarithmic*(*variants=defaultvariants, extendtick=0, extendlabel=None, autoexponent=10, epsilon=1e-10*)

class *graph.axis.parter.autolog*(...)

Evaluates candidate logarithmic tick and label patterns from *variants* (ordered from densest to sparsest) for rating by the axis rater.

10.6 Module `graph.axis.texter`: Axis Texters

Texters format the numeric value of each `tick` whose `labellevel` is not `None` and whose `label` is not already set manually. All texters are callable so you can clone and tweak them via keyword arguments.

```
class graph.axis.texter.default(multiplicationtex='\cdot', multiplicationunicode='.', base=Fraction(10),
                                skipmantissaunity=skipmantissaunity.all, minusunity='-', minexponent=4,
                                minnegexponent=None, uniformexponent=True, mantissatexter=decimal(),
                                basetexter=decimal(), exponenttexter=decimal(),
                                labelattrs=[text.mathmode], **kwargs)
```

The default texter for linear and logarithmic axes. Formats moderate numbers in plain decimal notation and automatically switches to scientific notation ($m \cdot 10^e$) for large or small exponents, producing a `text.MultiEngineText` that works seamlessly with both TeX/LaTeX engines and `text.UnicodeEngine`.

- `multiplicationtex`, `multiplicationunicode`: Multiplication symbol placed between the mantissa and base for TeX engines and `text.UnicodeEngine`, respectively.
- `base`: Exponent base (default `Fraction(10)`).
- `skipmantissaunity`: Controls when a mantissa of 1 is omitted (so $1 \cdot 10^5$ is written simply as 10^5):
 - `skipmantissaunity.never` (0): Always write $1 \cdot 10^e$.
 - `skipmantissaunity.each` (1): Omit 1 on any individual tick whose mantissa is unity.
 - `skipmantissaunity.all` (2, default): Omit 1 only when *all* exponentially formatted ticks on the axis have a mantissa of unity (as on a standard logarithmic axis).
- `minexponent`, `minnegexponent`: Minimum positive exponent (default 4, i.e. 10^4) and minimum negative exponent magnitude (defaults to `minexponent`, i.e. 10^{-4}) at which exponential notation is used.
- `uniformexponent`: When `True` (default), if any non-zero tick on the axis triggers exponential notation, all non-zero ticks on the axis are formatted in exponential notation for visual consistency.
- `mantissatexter`, `basetexter`, `exponenttexter`: `decimal` texters used to format the mantissa, base, and exponent (can also be configured via underscore-split keywords such as `mantissatexter_equalprecision=True`).
- `labelattrs`: Text attributes added to each formatted tick (default `[text.mathmode]`).

```
class graph.axis.texter.decimal(prefix="", infix="", suffix="", equalprecision=False, decimalsep='.',
                                thousandsep="", thousandthpartsep="", plus="", minus="-",
                                period='\overline{%s}', labelattrs=[text.mathmode])
```

Formats ticks in fixed-point decimal notation.

- `prefix`, `infix`, `suffix`: Strings inserted before the sign, immediately after the sign, and at the end of the number, respectively.
- `equalprecision`: When `True`, pads all decimal tick labels on the axis with trailing zeros to match the tick with the most decimal places.
- `decimalsep`, `thousandsep`, `thousandthpartsep`: Separators for the decimal point and 3-digit groups in the integer and fractional parts.
- `plus`, `minus`: Sign strings for non-negative and negative values.
- `period`: Format string with a single `%s` placeholder used to render repeating decimal periods (default `r"\overline{%s}"`).

```
class graph.axis.texter.rational(prefix="", infix="", suffix="", numprefix="", numinfix="", numsuffix="",
                                denomprefix="", denominfix="", denomsuffix="", plus="", minus='-',
                                minuspos=0, over='{{%s}\\over{%s}}', equaldenom=False, skip1=True,
                                skipnum0=True, skipnum1=True, skipdenom1=True,
                                labelattrs=[text.mathmode])
```

Formats ticks as exact reduced fractions $\frac{a}{b}$ (especially useful for axes partitioned in multiples of π or rational fractions).

- *prefix, infix, suffix*: Strings placed around the entire fraction.
- *numprefix, numinfix, numsuffix*: Strings placed around the numerator (for example, `numsuffix=r"\pi"`).
- *denomprefix, denominfix, denomsuffix*: Strings placed around the denominator.
- *minuspos*: Placement of the sign: 0 (in front of the fraction), 1 (in the numerator), or -1 (in the denominator).
- *over*: Format string with two %s placeholders for numerator and denominator (default `r"{{%s}}\over{%s}"`; use `r"{{%s}}/{%s}"` for inline slash fractions).
- *equaldenom*: When True, expresses all fractions on the axis over their least common denominator instead of reducing each fraction individually.
- *skip1, skipnum0, skipnum1, skipdenom1*: Booleans controlling automatic simplification when the fraction is ± 1 , the numerator is 0 or ± 1 , or the denominator is 1.

10.7 Module *graph.axis.painter*: Axis Painters

Painters draw the visual representation of an axis onto an `axiscanvas`.

10.7.1 Helper Classes (*rotatetext* and *ticklength*)

```
class graph.axis.painter.rotatetext(direction, epsilon=1e-10)
```

Computes a *trafo.rotate* transformation for axis labels or titles at angle *direction* (in degrees) relative to the inward tick direction, automatically flipping the text by 180° whenever it would otherwise appear upside-down.

parallel

`rotatetext(90)` — orients text parallel to the axis baseline.

orthogonal

`rotatetext(180)` — orients text perpendicular to the axis baseline.

```
class graph.axis.painter.ticklength(initial, factor)
```

A changeable attribute (*attr.changeattr*) that generates a geometric progression of visual lengths $\text{initial} \cdot \text{factor}^k$ for major ticks ($k = 0$), subticks ($k = 1$), sub-subticks ($k = 2$), etc.

Predefined instances with `factor = 1 / goldenmean` and base length `0.12 * unit.v_cm` scaled by powers of $\sqrt{2}$ are provided as class attributes: `ticklength.SHORT`, `ticklength.SHORTt`, `ticklength.SHOrt`, `ticklength.SHort`, `ticklength.Short`, `ticklength.short`, `ticklength.normal`, `ticklength.long`, `ticklength.Long`, `ticklength.Longt`, `ticklength.LONG`, and `ticklength.LONGt`.

10.7.2 Painter Classes

```
class graph.axis.painter.regular(innerticklength=ticklength.normal, outerticklength=None, tickattrs=[],
                                gridattrs=None, basepathattrs=[], labeldist=0.3 * unit.v_cm,
                                labelattrs=[], labeldirection=None, labelhequalize=0, labelvequalize=1,
                                titledist=0.3 * unit.v_cm, titleattrs=[], titledirection=rotatetext.parallel,
                                titlepos=0.5, textengine=None)
```

Default painter for continuous axes (*linear*, *logarithmic*, *timeaxis*).

- *innerticklength*, *outerticklength*: Visual length (or changeable *ticklength*) of ticks pointing inside and outside the graph, or *None* to omit inner or outer ticks.
- *tickattrs*: Stroke attributes for ticks (supports changeable attributes to style major ticks and subticks differently), or *None* to suppress ticks.
- *gridattrs*: Stroke attributes for grid lines across the graph (for example, `[color.gray(0.8)]` or `[attr.changelist([style.linestyle.solid, style.linestyle.dotted])]`), or *None* (default) to disable grid lines.
- *basepathattrs*: Stroke attributes for the axis baseline (merged with `[style.linecap.square]`), or *None* to omit the baseline.
- *labeldist*: Visual distance between the axis baseline (or outer ticks) and the tick labels (default `0.3 * unit.v_cm`).
- *labelattrs*: Text attributes for tick labels (merged with `[text.halign.center, text.vshift.mathaxis]`), or *None* to suppress tick labels.
- *labeldirection*: Optional *rotatetext* instance to rotate tick labels relative to the axis.
- *labelhequalize*, *labelvequalize*: Aligns all tick labels along a common horizontal or vertical line on straight vertical or horizontal axes.
- *titledist*, *titleattrs*, *titledirection*, *titlepos*: Distance, text attributes (or *None* to hide), rotation (*rotatetext.parallel* by default), and normalized position (0.5 = centered) of the axis title.
- *textengine*: Optional text engine override (defaults to the graph's text engine).

```
class graph.axis.painter.linked(innerticklength=ticklength.short, outerticklength=None, tickattrs=[],
                                gridattrs=None, basepathattrs=[], labeldist=0.3 * unit.v_cm,
                                labelattrs=None, labeldirection=None, labelhequalize=0,
                                labelvequalize=1, titledist=0.3 * unit.v_cm, titleattrs=None,
                                titledirection=rotatetext.parallel, titlepos=0.5, textengine=None)
```

Subclass of *regular* used by default for opposite linked axes (x2, y2), with *innerticklength=ticklength.short* and labels and titles disabled (*labelattrs=None*, *titleattrs=None*).

```
class graph.axis.painter.bar(innerticklength=None, outerticklength=None, tickattrs=[], basepathattrs=[],
                             namedist=0.3 * unit.v_cm, nameattrs=[], namedirection=None, namepos=0.5,
                             namehequalize=0, namevequalize=1, titledist=0.3 * unit.v_cm, titleattrs=[],
                             titledirection=rotatetext.parallel, titlepos=0.5, textengine=None)
```

```
class graph.axis.painter.linkedbar(...)
```

Painters for *bar* and *nestedbar* axes. The *name** parameters control the placement and formatting of the categorical bar labels, while *innerticklength* and *outerticklength* optionally mark the boundaries of each bar subaxis. *linkedbar* disables category names and the axis title by default.

```
class graph.axis.painter.split(breaklinesdist=0.05 * unit.v_cm, breaklineslength=0.5 * unit.v_cm,
                               breaklinesangle=-60, titledist=0.3 * unit.v_cm, titleattrs=[],
                               titledirection=rotatetext.parallel, titlepos=0.5, textengine=None)
```

```
class graph.axis.painter.linkedsplit(...)
```

Painters for *split* axes, drawing parallel break slashes separated by *breaklinesdist* of length *breaklineslength* at angle *breaklinesangle* (in degrees) between adjacent subaxes.

10.8 Module *graph.axis.rater*: Axis Raters

When an automatic partitioner (*autolinear* or *autologarithmic*) produces multiple candidate tick lists, an axis *rater* assigns a non-negative penalty score to each candidate (where smaller scores are better):

1. **Tick count and range rating:** Scores the number of ticks, subticks, and labels against target optima (*cube*) and penalizes excessive range expansion beyond the data bounds.
2. **Label layout rating:** For the top-ranked candidates, paints the axis and checks the spacing between adjacent label boxes (*distance*). Any candidate whose labels overlap is rejected completely.

class `graph.axis.rater.cube`(*opt*, *left*=None, *right*=None, *weight*=1)

Cubic penalty function $\text{weight} \cdot ((x - \text{opt}) / (\text{other} - \text{opt}))^3$ evaluating how far a count x deviates from the optimal value *opt*, where *other* is *left* (default 0) for $x < \text{opt}$ and *right* (default $3 * \text{opt}$) for $x > \text{opt}$.

class `graph.axis.rater.distance`(*opt*=1 * *unit.v_cm*, *weight*=0.1)

Rates the visual separation between adjacent tick label boxes against the optimal separation *opt* (rejecting negative separations where labels overlap).

class `graph.axis.rater.rater`(*ticks*, *labels*, *range*, *distance*=*distance*(1 * *unit.v_cm*), *density*=1)

Combines lists of *cube* raters for *ticks* and *labels* at each hierarchy level, a *cube* rater for *range* extension, and a *distance* rater for label spacing. Increasing *density* (e.g. *density*=1.5 or 2) scales the optimal tick and label counts upward to favor denser ticks; decreasing *density* (e.g. *density*=0.7) favors fewer ticks.

class `graph.axis.rater.linear`(*ticks*=[*cube*(4), *cube*(10, *weight*=0.5)], *labels*=[*cube*(4)], *range*=*cube*(1, *weight*=2), ***kwargs*)

class `graph.axis.rater.lin`(...)

Default rater for linear axes.

class `graph.axis.rater.logarithmic`(*ticks*=[*cube*(5, *right*=20), *cube*(20, *right*=100, *weight*=0.5)], *labels*=[*cube*(5, *right*=20), *cube*(5, *right*=20, *weight*=0.5)], *range*=*cube*(1, *weight*=2), ***kwargs*)

class `graph.axis.rater.log`(...)

Default rater for logarithmic axes.

10.9 Module *graph.axis.positioner*: Axis Positioners

Positioner instances define the geometric placement of an axis on a canvas and are created automatically by `graph`, `graphxy`, `graph.graphxyz`, and `graph.axis.pathaxis()`. Every positioner implements the following interface:

class `graph.axis.positioner._positioner`

vbasepath(*v1*=None, *v2*=None)

Returns the *path.path* of the axis baseline between normalized graph coordinates *v1* and *v2*.

vgridpath(*v*)

Returns the grid line *path.path* at normalized coordinate *v*, or None if no grid is defined.

vtickpoint_pt(*v*)

Returns (*x_pt*, *y_pt*) in PostScript points on the baseline at normalized coordinate *v*.

vtickdirection(*v*)

Returns the unit normal vector (*dx*, *dy*) pointing toward the graph interior at normalized coordinate *v*.

MODULE *BOX*: CONVEX BOX HANDLING

The *box* module provides convex polygon (*polygon*) and rectangular (*rect*) box classes with an associated reference point (*center*). Boxes are used throughout PyX to represent the outer boundary and anchor point of text labels (*text*, *textbox*), align multiple objects along lines or circles, tile objects with uniform spacing, and compute boundary intersections for *connector* paths.

All direction arguments (*dx*, *dy*) in *box* are 2D direction vectors where at least one of *dx* or *dy* must be non-zero (their length is normalized automatically).

11.1 Class *polygon*

class *box.polygon*(*corners=None, center=None*)

Represents a convex polygon defined by a sequence of (*x*, *y*) vertex tuples in *corners* ordered counter-clockwise, and an optional reference point (*x*, *y*) in *center* (in user units or PyX lengths). If *center* is *None*, the arithmetic mean (centroid) of *corners* is used.

A point-based base class *polygon_pt* takes all coordinates in PostScript points.

path(*centerradius=None*)

Returns a closed *path.path* tracing the polygon's perimeter. If *centerradius* is given, an additional small circle of radius *centerradius* is included at the box's center.

Tip

To round the corners of a box path, apply *deformer.smoothed* to the returned path—for example, *deformer.smoothed(0.2).deform(mybox.path())*.

transform(**trafos, keep_anchor=False*)

Applies one or more *trafo.trafo* transformations in place to the polygon's corners (and to *center*, unless *keep_anchor=True*).

reltransform(**trafos*)

Applies one or more transformations in place relative to the box's *center* (translating *center* to the origin, applying *trafos*, and translating back).

circlealignvector(*a, dx, dy*)

circlealignvector_pt(*a_pt, dx, dy*)

Returns the displacement vector (*vx*, *vy*) required to move the box along the direction (*dx*, *dy*) so that its boundary touches a circle of radius *a* centered at the box's current *center* (see figure *Aligning boxes on a circle* (*circlealign()*) and along a line (*linealign()*) for identical direction vectors and distances.).

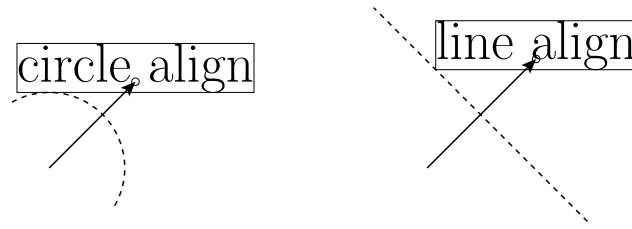


Fig. 1: Aligning boxes on a circle (*circlealign()*) and along a line (*linealign()*) for identical direction vectors and distances.

linealignvector(*a*, *dx*, *dy*)

linealignvector_pt(*a_pt*, *dx*, *dy*)

Returns the displacement vector (*vx*, *vy*) required to move the box along the direction (*dx*, *dy*) so that its boundary touches a line perpendicular to (*dx*, *dy*) at distance *a* from the box's current center (see figure *Aligning boxes on a circle (circlealign()) and along a line (linealign()) for identical direction vectors and distances.*).

circlealign(*a*, *dx*, *dy*)

circlealign_pt(*a_pt*, *dx*, *dy*)

Translates the box in place by *circlealignvector()* and returns *self*.

linealign(*a*, *dx*, *dy*)

linealign_pt(*a_pt*, *dx*, *dy*)

Translates the box in place by *linealignvector()* and returns *self*.

extent(*dx*, *dy*)

extent_pt(*dx*, *dy*)

Returns the total projected width of the box along the direction (*dx*, *dy*).

pointdistance(*x*, *y*)

pointdistance_pt(*x_pt*, *y_pt*)

Returns the shortest Euclidean distance from an exterior point (*x*, *y*) to the boundary of the box.

boxdistance(*other*)

boxdistance_pt(*other*, *epsilon=1e-10*)

Returns the shortest Euclidean distance between *self* and another box *other*. Raises *BoxCrossError* if the two boxes intersect.

bbox()

Returns the axis-aligned *bbox.bbox_pt* enclosing the box.

exception *box.BoxCrossError*

Raised by *polygon.boxdistance()* when two boxes have intersecting edges.

11.2 Aligning and Tiling Lists of Boxes

box.circlealignequal(*polygons*, *a*, *dx*, *dy*)

`box.circleanequal_pt(polygons, a_pt, dx, dy)`

Computes the `circleanvector()` for each box in *polygons* and translates all boxes by the longest of those vectors.

`box.linealignequal(polygons, a, dx, dy)`

`box.linealignequal_pt(polygons, a_pt, dx, dy)`

Computes the `linealignvector()` for each box in *polygons* and translates all boxes by the longest of those vectors. Axis painters use this function when `labelvequalize` or `labelhequalize` is enabled so that all tick labels share a uniform baseline alignment.

`box.tile(polygons, a, dx, dy)`

`box.tile_pt(polygons, a_pt, dx, dy)`

Translates the boxes in *polygons* sequentially along (dx, dy) with step size equal to the maximum box extent in direction (dx, dy) plus separation *a*, and returns the step size.

11.3 Class `rect`

class `box.rect(x, y, width, height, relcenter=(0, 0), abscenter=(0, 0))`

Convenience subclass of `polygon` representing an axis-aligned rectangle with lower-left corner (x, y) and dimensions *width* and *height* (in user units or PyX lengths). A point-based variant `rect_pt` takes coordinates in PostScript points.

The reference center of the rectangle is placed at

$$(x + \text{relcenter}_x \cdot \text{width} + \text{abscenter}_x, y + \text{relcenter}_y \cdot \text{height} + \text{abscenter}_y),$$

which defaults to the lower-left corner (x, y) when `relcenter=(0, 0)` and `abscenter=(0, 0)`. Set `relcenter=(0.5, 0.5)` to place the anchor at the geometric center of the rectangle.

MODULE CONNECTOR

The *connector* module constructs paths (*normpath.normpath* subclasses) that connect two *box.polygon* instances (such as *text.texttextbox_pt* / *text.unicodetextbox_pt* objects or *box.rect* boxes) using straight lines, circular arcs, cubic Bézier curves, or two-segment polylines.

12.1 Geometry, Angles, and Bulges

All connector classes take two box instances (*box1* and *box2*) as their first two arguments and construct a path leading from *box1* to *box2*:

1. The geometric curve is constructed between the centers (*box1.center* and *box2.center*) of the two boxes.
2. The path is then trimmed at its intersections with the boundary outlines of *box1* and *box2*.
3. Finally, the trimmed path can be shortened at its start and end by the distances specified in *boxdists* (a 2-element sequence [*dist1*, *dist2*], defaulting to [*0*, *0*]).

```
from pyx import *

c = canvas.canvas()
b1 = c.text(0, 0, "Start", [text.halign.center, text.vshift.mathaxis])
b2 = c.text(4, 1, "End", [text.halign.center, text.vshift.mathaxis])

c.stroke(connector.arc(b1, b2, boxdists=[0.2, 0.2]), [deco.earrow.normal])
```

12.1.1 Angles and Bulge Parameters

Connectors that bend (*arc*, *curve*, and *twolines*) are parameterized by tangent angles and/or bulge distances (illustrated in figure *Geometric meaning of the angle and bulge parameters for connector.arc (left) and connector.curve (right)*):

- **Relative vs. absolute angles:** Relative angles (*relangle*, *relangle1*, *relangle2*) are measured in degrees relative to the straight line connecting *box1.center* to *box2.center*. Absolute angles (*absangle1*, *absangle2*) are measured counter-clockwise from the positive *x*-axis. Whenever both an absolute and a relative angle are specified for the same endpoint, the absolute angle takes precedence.
- **Bulge distances:** Bulge parameters (*relbulge* and *absbulge*) control how far the connector deviates from the straight line between the box centers. *relbulge* is a dimensionless fraction of the center-to-center distance, whereas *absbulge* is an absolute PyX length. If both *relbulge* and *absbulge* are given, their contributions are summed.

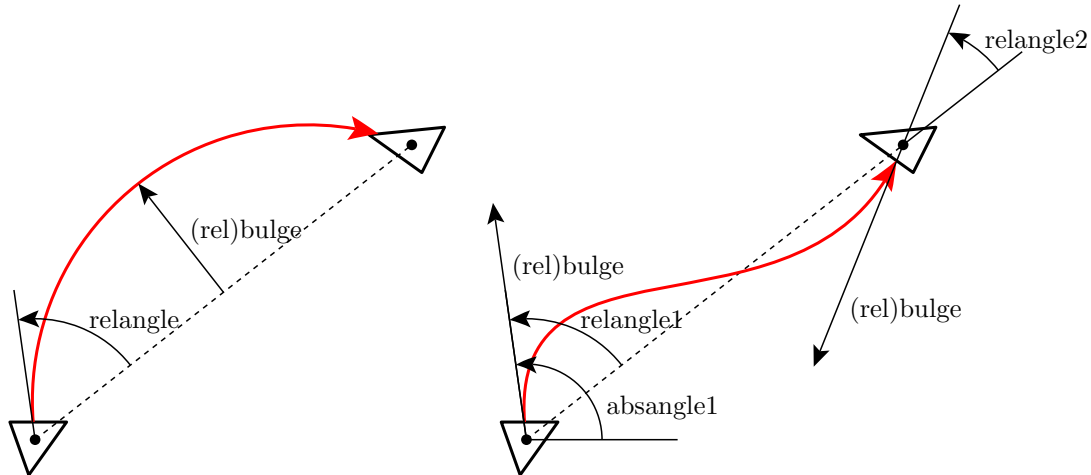


Fig. 1: Geometric meaning of the angle and bulge parameters for `connector.arc` (left) and `connector.curve` (right).

12.2 Connector Classes

Each connector class is available both in user units (e.g. `line`, `arc`, `curve`, `twolines`) and in PostScript points (with a `_pt` suffix, where `boxdists` and `absbulge` are given in points).

class `connector.line(box1, box2, boxdists=(0, 0))`

Connects `box1` and `box2` with a straight line segment between their centers, trimmed at the box outlines and shortened by `boxdists`.

class `connector.arc(box1, box2, relangle=45, absbulge=None, relbulge=None, boxdists=[0, 0])`

Connects `box1` and `box2` with a circular arc.

- *relangle*: Angle (in degrees) between the chord connecting the box centers and the arc's tangent at the centers (default 45).
- *absbulge*, *relbulge*: Maximum perpendicular distance (sagitta) between the arc and the straight chord connecting the box centers. Specifying either *absbulge* or *relbulge* overrides *relangle*.

class `connector.curve(box1, box2, relangle1=45, relangle2=45, absangle1=None, absangle2=None, absbulge=0, relbulge=0.39, boxdists=[0, 0])`

Connects `box1` and `box2` with a cubic Bézier curve. Here the bulge specifies the distance from each box center to the corresponding Bézier control point along the departure/arrival tangent direction (see the right panel of figure *Geometric meaning of the angle and bulge parameters for connector.arc (left) and connector.curve (right)*. for sign conventions).

- *relangle1*, *relangle2*: Departure and arrival tangent angles relative to the center-to-center line (default 45 for both).
- *absangle1*, *absangle2*: Absolute departure and arrival tangent angles in degrees (override *relangle1* / *relangle2* if given).
- *absbulge*, *relbulge*: Control-point distances (default *relbulge*=0.39, which closely approximates a circular arc when *relangle1* = *relangle2* = 45).

class `connector.twolines(box1, box2, absangle1=None, absangle2=None, relangle1=None, relangle2=None, relangleM=None, length1=None, length2=None, bezierradius=None, beziersoftness=1, arcradius=None, boxdists=[0, 0])`

Connects `box1` and `box2` with a two-segment polyline meeting at an intermediate corner point. You must supply

an unambiguous combination of two constraints chosen from the departure angle (*absangle1* or *relangle1*), arrival angle (*absangle2* or *relangle2*), interior angle between the two segments (*relangleM*), first segment length (*length1*), or second segment length (*length2*).

MODULE *EPSFILE*: EPS FILE INCLUSION

The *epsfile* module allows you to embed external Encapsulated PostScript (EPS) files into a PyX *canvas.canvas*, with full control over position, scaling, alignment, and clipping:

```
from pyx import *

c = canvas.canvas()
c.insert(epsfile.epsfile(0, 0, "figure.eps", width=6, align="cc"))
c.writeEPSfile("output")
```

13.1 Class *epsfile*

class *epsfile.epsfile*(*x*, *y*, *filename*, *width=None*, *height=None*, *scale=None*, *align='bl'*, *clip=1*,
translatebbox=True, *bbox=None*, *kpsearch=False*)

Creates a canvas item representing the external EPS file *filename* placed at (*x*, *y*) (in user units or PyX lengths).

Argument	Description
<i>x</i> , <i>y</i>	Coordinates of the reference point at which the EPS graphic is placed.
<i>filename</i>	Path to the EPS file.
<i>width=None</i>	Desired width of the graphic, or <i>None</i> to use its natural bounding-box width (or scale proportionally from <i>height</i>). Cannot be combined with <i>scale</i> .
<i>height=None</i>	Desired height of the graphic, or <i>None</i> to use its natural bounding-box height (or scale proportionally from <i>width</i>). Cannot be combined with <i>scale</i> .
<i>scale=None</i>	Uniform scaling factor, or <i>None</i> for no scaling. Cannot be combined with <i>width</i> or <i>height</i> .
<i>align="bl"</i>	Two-character alignment string specifying how the graphic is aligned relative to (<i>x</i> , <i>y</i>). The first character sets vertical alignment ("b" bottom, "c" center, "t" top) and the second sets horizontal alignment ("l" left, "c" center, "r" right).
<i>clip=1</i>	When true (default), clips the EPS graphic to its bounding box.
<i>translatebbox</i>	When true (default), translates the EPS graphic so that the lower-left corner of its bounding box sits at the reference origin before scaling and alignment.
<i>bbox=None</i>	Optional <i>bbox.bbox</i> overriding the %%BoundingBox / %%HiResBoundingBox comments read from the EPS file.
<i>kpsearch</i>	When true, locates <i>filename</i> via the TeX kpathsea file search system (<i>config</i>).

13.1.1 Output Format Compatibility

- **PostScript and EPS output:** The EPS file is embedded directly as vector PostScript inside a `BeginDocument` / `EndDocument` wrapper.
- **PDF output:** Because PDF viewers cannot execute arbitrary PostScript code, PyX automatically rasterizes the included EPS file to a PNG bitmap using Ghostscript (`canvas.canvas.pipeGS()`) when writing PDF output. To include vector graphics natively in PDF output, convert the file to PDF and use the [*pdffile*](#) module instead.

MODULE *PDFFILE*: PDF FILE INCLUSION

The *pdffile* module embeds pages from external PDF files directly as native vector PDF Form XObjects when generating PDF output, preserving all vector paths, embedded fonts, images, and resources:

```
from pyx import *

c = canvas.canvas()
c.insert(pdffile.pdfFile(0, 0, "diagram.pdf", page=1, width=8))
c.writePDFfile("output")
```

14.1 Class *pdffile*

class *pdffile.pdfFile*(*x*, *y*, *filename*, *width=None*, *height=None*, *scale=None*, *align='bl'*, *clip=1*,
translatebbox=True, *bbbox=None*, *page=1*, *kpathsearch=False*)

Creates a canvas item representing page *page* of the external PDF file *filename* placed at (*x*, *y*) (in user units or PyX lengths).

Argument	Description
<i>x</i> , <i>y</i>	Coordinates of the reference point at which the PDF page is placed.
<i>filename</i>	Path to the PDF file, or an open binary file-like object.
<i>width=None</i>	Desired width of the graphic, or <i>None</i> to use its natural bounding-box width (or scale proportionally from <i>height</i>). Cannot be combined with <i>scale</i> .
<i>height=None</i>	Desired height of the graphic, or <i>None</i> to use its natural bounding-box height (or scale proportionally from <i>width</i>). Cannot be combined with <i>scale</i> .
<i>scale=None</i>	Uniform scaling factor, or <i>None</i> for no scaling. Cannot be combined with <i>width</i> or <i>height</i> .
<i>align="bl"</i>	Two-character alignment string specifying how the PDF page is aligned relative to (<i>x</i> , <i>y</i>). The first character sets vertical alignment ("b" bottom, "c" center, "t" top) and the second sets horizontal alignment ("l" left, "c" center, "r" right).
<i>clip=1</i>	When true (default), clips the included page to its bounding box.
<i>translatebbox=True</i>	When true (default), translates the PDF page so that the lower-left corner of its bounding box sits at the reference origin before scaling and alignment.
<i>bbbox=None</i>	Optional <i>bbbox</i> overriding the page's CropBox (or MediaBox if no CropBox is defined).
<i>page=1</i>	1-based page number of the PDF file to include.
<i>kpathsearch</i>	When true, locates <i>filename</i> via the TeX kpathsea file search system (<i>config</i>).

Note

Because *pdffile* embeds the page's content stream and resource dictionary directly as a PDF Form XObject, it is supported when writing PDF output (`canvas.canvas.writePDFfile()` and `document.document.writePDFfile()`).

MODULE *SVGFILE*: SVG FILE INCLUSION

The *svgfile* module allows you to include external Scalable Vector Graphics (SVG) files in a PyX *canvas.canvas*, either by embedding the XML tree directly in SVG output or by parsing the SVG vector paths and styles into a native PyX canvas that can be exported to PostScript, EPS, PDF, and SVG:

```
from pyx import *

c = canvas.canvas()
c.insert(svgfile.svgfile(0, 0, "icon.svg", width=4, parsed=True))
c.writePDFfile("output")
```

15.1 Class *svgfile*

class *svgfile.svgfile*(*x*, *y*, *filename*, *width=None*, *height=None*, *ratio=None*, *parsed=False*, *resolution=96*)

class *svgfile.svgfile_pt*(*x_pt*, *y_pt*, *filename*, *width_pt=None*, *height_pt=None*, *ratio=None*, *parsed=False*, *resolution=96*)

Creates a canvas item representing the SVG file *filename* placed with its lower-left corner at (*x*, *y*) (in user units or PyX lengths, or in PostScript points for *svgfile_pt*).

Argument	Description
<i>x</i> , <i>y</i>	Coordinates of the lower-left corner at which the SVG graphic is placed.
<i>filename</i>	Path to the SVG file.
<i>width=None</i>	Desired width of the graphic, or <i>None</i> to use its original width (or compute from <i>height</i> and <i>ratio</i>).
<i>height=None</i>	Desired height of the graphic, or <i>None</i> to use its original height (or compute from <i>width</i> and <i>ratio</i>).
<i>ratio=None</i>	Aspect ratio <i>width</i> / <i>height</i> used when only <i>width</i> or only <i>height</i> is specified. If <i>None</i> , the SVG file's original aspect ratio is preserved.
<i>parsed=False</i>	When <i>False</i> (default), embeds the raw SVG XML into SVG output. When <i>True</i> , parses the SVG elements into a native PyX <i>canvas.canvas</i> (stored in <i>self.canvas</i>).
<i>resolution=96</i>	Assumed resolution in dots per inch (dpi) used to convert unitless SVG user coordinates (pixels) to physical units.

15.2 Unparsed vs. Parsed Mode

15.2.1 Unparsed Mode (`parsed=False`)

In unparsed mode, PyX reads only the root `<svg>` attributes (`width`, `height`, and `viewBox`) to determine the graphic's dimensions and copies the SVG XML content directly into the output stream when writing an SVG file (`canvas.canvas.writeSVGfile()`). Unparsed mode supports the full SVG specification (including filters, gradients, and CSS), but can only be written to SVG output.

15.2.2 Parsed Mode (`parsed=True`)

In parsed mode, PyX parses the SVG document into native PyX paths, styles, transformations, and nested canvases stored in the instance's `canvas` attribute. The parser supports `<g>`, `<svg>`, `<path>` (including full SVG path data commands M, L, H, V, C, S, Q, T, A, Z), basic shapes (`<rect>`, `<circle>`, `<ellipse>`, `<line>`, `<polyline>`, `<polygon>`), coordinate transform attributes, and stroke and fill styles.

Parsed mode offers two major advantages:

1. **Cross-format output:** The included SVG graphic can be written to PostScript, EPS, PDF, and SVG alike.
2. **Direct path access:** Because `self.canvas` is a standard PyX `canvas.canvas`, you can inspect or manipulate the parsed paths and sub-canvases directly in Python.

15.3 Coordinate Resolution (`resolution`)

Although SVG is a vector format, SVG files frequently specify `width`, `height`, or coordinates in unitless user coordinates (`px`) rather than physical units (`pt`, `mm`, `cm`, `in`). The `resolution` parameter defines how many SVG user pixels correspond to one physical inch (72 PostScript points):

- **96 dpi** (the default): Used by CSS, modern web browsers, and Inkscape 0.92+.
- **90 dpi**: Used by the original W3C SVG 1.1 specification and older versions of Inkscape.
- **72 dpi**: Used by Adobe Illustrator (where 1 user unit equals 1 PostScript point).

When an SVG file specifies physical units on its root `width` and `height` (as PyX's own SVG writer does) and `parsed=True` is used, the result is completely independent of *resolution*.

BITMAPS

16.1 Introduction

Although PyX is primarily designed for scalable vector graphics, the *bitmap* module provides full support for embedding raster (bitmap) images in PostScript, EPS, PDF, and SVG output. You can construct images directly from raw byte buffers in Python, load JPEG files natively via *jpegimage*, or pass any *Pillow* (PIL) Image object directly to *bitmap*.

16.1.1 Example: Creating a Bitmap from Raw Bytes

To create a bitmap from raw binary pixel data, first wrap the byte buffer in an *image* instance and then place it at a canvas position with *bitmap*:

```
from pyx import *

image_bw = bitmap.image(2, 2, "L", b"\0\377\377\0")
image_rgb = bitmap.image(3, 2, "RGB", b"\77\77\77\177\177\177\277\277\277"
                                b"\377\0\0\0\377\0\0\0\377")

bitmap_bw = bitmap.bitmap(0, 1, image_bw, height=0.8)
bitmap_rgb = bitmap.bitmap(0, 0, image_rgb, height=0.8)

c = canvas.canvas()
c.insert(bitmap_bw)
c.insert(bitmap_rgb)
c.writeEPSfile("bitmap")
```

Here *image_bw* is a 2×2 8-bit grayscale ("L") image containing two black (0) and two white (255) pixels, ordered row by row from the top-left corner. *image_rgb* is a 3×2 RGB image whose top row contains three shades of gray and whose bottom row contains red, green, and blue pixels. Because raw *image* buffers do not carry DPI metadata, we specify *height=0.8* when constructing the *bitmap* canvas items. Figure *Placing grayscale and RGB bitmap instances on a canvas* shows the resulting output.



Fig. 1: Placing grayscale and RGB *bitmap* instances on a canvas.

16.2 Image Containers and File Readers

class `bitmap.image`(*width, height, mode, data, compressed=None, palette=None*)

Container for 8-bit-per-channel pixel data.

- *width, height*: Image dimensions in pixels.
- *mode*: Color mode string. Supported modes are:
 - "L" (8-bit grayscale), "RGB" (24-bit RGB), "CMYK" (32-bit CMYK)
 - Modes with an 8-bit alpha channel: "LA", "RGBA", "CMYKA" (trailing alpha) or "AL", "ARGB", "ACMYK" (leading alpha)
 - "P" (8-bit palette-indexed color, requiring a [palette](#) instance in *palette*)
- *data*: A bytes object containing the pixel values row by row from top left to bottom right. For uncompressed data, `len(data)` must equal `len(mode) * width * height`.
- *compressed*: `None` for uncompressed pixel bytes, or "Flate" / "DCT" if *data* is already Flate- or JPEG-compressed.
- *palette*: Optional [palette](#) instance when *mode*="P".

class `bitmap.palette`(*mode, data*)

Container for palette color tables used by indexed-color (*mode*="P") [image](#) instances, where *mode* is "L", "RGB", or "CMYK" and *data* is a bytes object containing the palette entries.

class `bitmap.jpegimage`(*file*)

Reads a JPEG/JFIF image from *file* (either a filename string or an open binary file-like object). [jpegimage](#) parses the JPEG headers to extract the pixel dimensions, color components ("L", "RGB", or "CMYK"), and embedded DPI resolution (if present), and embeds the DCT-compressed stream directly into the output without decoding and re-encoding—preserving 100% of the original image quality and minimizing file size.

16.3 Placing Bitmaps on a Canvas

class `bitmap.bitmap`(*xpos, ypos, image, width=None, height=None, ratio=None, PSstoreimage=0, PSmaxstrlen=4093, PSbinexpand=1, compressmode='Flate', flatecompresslevel=6, dctquality=75, dctoptimize=0, dctprogression=0*)

Creates a canvas item that places *image* with its lower-left corner at (*xpos*, *ypos*) (in user units or PyX lengths). A point-based variant `bitmap_pt` takes *xpos_pt*, *ypos_pt*, *width_pt*, and *height_pt* in PostScript points, and the lower-level base class `bitmap_trafo` accepts an arbitrary `trafo.trafo_pt` mapping the unit square to the target image rectangle.

- *image*: An instance of [image](#), [jpegimage](#), or any Pillow (PIL .Image) object.
- *width, height, ratio*: Target physical dimensions of the bitmap on the canvas. If *image* does not provide DPI metadata (in `image.info["dpi"]`), at least *width* or *height* must be specified. When only one dimension is given, the other is computed from *ratio* (or from the pixel aspect ratio if *ratio*=`None`).
- *PSstoreimage*: When true, stores the compressed image data in PostScript interpreter memory as a reusable resource rather than streaming it inline. This avoids duplicating image bytes when the same bitmap is drawn multiple times in PostScript output.
- *PSmaxstrlen*: Maximum PostScript string length (default 4093) used when *PSstoreimage* is enabled.
- *PSbinexpand*: Expansion factor (1 or 2) used when converting low-bit-depth PIL images for PostScript output.

- *compressmode*: Stream compression method used in PostScript and PDF output: "Flate" (lossless zlib compression, the default for uncompressed images), "DCT" (JPEG compression, supported for *jpegimage* and Pillow images), or None (uncompressed). When *image* is already compressed (such as a *jpegimage*), *compressmode* is automatically set to None so the compressed stream is passed through untouched.
- *flatecompresslevel*: Zlib compression level (1–9, default 6) when *compressmode*="Flate".
- *dctquality*, *dctoptimize*, *dctprogression*: JPEG encoding parameters passed to Pillow when *compressmode*="DCT". Keep *dctprogression*=0 and *dctoptimize*=0 for maximum PostScript printer compatibility.

Note

- **Alpha channels (transparency)**: Images with an alpha channel ("LA", "RGBA", "CMYKA", etc.) are automatically split into a color image and a soft-mask (/SMask) in PDF output, or exported as RGBA PNG data in SVG output. Standard PostScript does not support soft masks and logs a warning while ignoring the alpha channel.
- **SVG output**: When writing SVG output, *bitmap* embeds the image as a base64-encoded `data:image/png` or `data:image/jpeg` URI inside an SVG `<image>` element.

MODULE *BBOX*

The *bbox* module defines axis-aligned bounding boxes (*bbox*, *bbox_pt*, and *empty*) used throughout PyX to compute the spatial extent of paths, text boxes, canvases, and pages. While you most commonly obtain a bounding box by calling an object's *bbox()* method (such as *canvas.canvas.bbox()* or *path.path.bbox()*), you can also construct and manipulate bounding boxes directly—for example, to pass a custom clipping or page bounding box to *canvas.canvas.writeEPSfile()* or *document.page*.

17.1 Constructing Bounding Boxes

class *bbox.bbox*(*llx*, *lly*, *urx*, *ury*)

Constructs a bounding box with lower-left corner (*llx*, *lly*) and upper-right corner (*urx*, *ury*) specified in user units or PyX lengths.

class *bbox.bbox_pt*(*llx_pt*, *lly_pt*, *urx_pt*, *ury_pt*)

Constructs a bounding box with coordinates (*llx_pt*, *lly_pt*) and (*urx_pt*, *ury_pt*) specified directly in PostScript points. All *bbox()* methods on PyX objects return *bbox_pt* instances.

class *bbox.empty*

Constructs an empty bounding box (containing no points, with *llx_pt*=None). An empty bounding box evaluates to False in a boolean context (*bool(bbox.empty()) == False*), whereas any non-empty bounding box evaluates to True.

17.2 Set Operations and Methods

Two bounding boxes *b1* and *b2* can be combined using arithmetic operators:

- **Union** (*b1* + *b2* or *b1* += *b2*): Returns the smallest bounding box enclosing both *b1* and *b2*. Adding an *empty* bounding box leaves the other bounding box unchanged.
- **Intersection** (*b1* * *b2* or *b1* *= *b2*): Returns the overlapping region of *b1* and *b2* (or an *empty* bounding box if either input is empty).

In addition, *bbox* and *bbox_pt* provide the following methods (where measurement methods without a *_pt* suffix return true PyX lengths *unit.t_pt*, and *_pt* variants take or return plain floats in PostScript points):

Method	Description
<code>copy()</code>	Returns a copy of the bounding box.
<code>set(other)</code>	Copies the coordinates of <i>other</i> into <i>self</i> .
<code>intersects(other)</code>	Returns True if <i>self</i> and <i>other</i> overlap (and neither is empty), or False/0 otherwise.
<code>includepoint_pt(x_pt, y_pt)</code>	Expands <i>self</i> in place to include the point (<i>x_pt</i> , <i>y_pt</i>).
<code>transform(trafo)</code>	Transforms the four corners of <i>self</i> in place by <i>trafo</i> and updates <i>self</i> to their enclosing axis-aligned box.
<code>transformed(trafo)</code>	Returns a new bounding box enclosing <i>self</i> after transformation by <i>trafo</i> .
<code>enlarge(all=0, bottom=None, left=None, top=None, right=None)</code>	Expands <i>self</i> in place on each side by the specified PyX lengths (using <i>all</i> for any side left as None). Also available as <code>enlarge_pt(...)</code> .
<code>enlarged(all=0, bottom=None, left=None, top=None, right=None)</code>	Returns a new bounding box expanded on each side by the specified amounts (using <i>all</i> for any side left as None). Also available as <code>enlarged_pt(...)</code> .
<code>path() / rect()</code>	Returns a rectangular <i>path.rect_pt</i> path tracing the boundary of the bounding box.
<code>width() / width_pt()</code>	Returns the horizontal width of the bounding box.
<code>height() / height_pt()</code>	Returns the vertical height of the bounding box.
<code>left() / left_pt()</code>	Returns the <i>x</i> -coordinate of the left edge (<i>llx</i>).
<code>right() / right_pt()</code>	Returns the <i>x</i> -coordinate of the right edge (<i>urx</i>).
<code>bottom() / bottom_pt()</code>	Returns the <i>y</i> -coordinate of the bottom edge (<i>lly</i>).
<code>top() / top_pt()</code>	Returns the <i>y</i> -coordinate of the top edge (<i>ury</i>).
<code>center() / center_pt()</code>	Returns the (<i>x</i> , <i>y</i>) coordinates of the center of the bounding box.
<code>lowrestuple_pt()</code>	Returns (<code>floor(llx_pt)</code> , <code>floor(lly_pt)</code> , <code>ceil(urx_pt)</code> , <code>ceil(ury_pt)</code>) as integer points (used for PostScript <code>%%BoundingBox</code>).
<code>highrestuple_pt()</code>	Returns (<i>llx_pt</i> , <i>lly_pt</i> , <i>urx_pt</i> , <i>ury_pt</i>) in exact floating-point points (used for <code>%%HiResBoundingBox</code>).

MODULE *COLOR*

The *color* module provides classes representing colors in four device color spaces (*gray*, *rgb*, *cmyk*, and *hsb*), continuous color gradients (*gradient*), and opacity attributes (*transparency*). Because all color and transparency instances act as both `style.strokeStyle` and `style.fillStyle` attributes, they can be passed directly to *canvas.canvas.stroke()*, *canvas.canvas.fill()*, *canvas.canvas.draw()*, *canvas.canvas.text()*, or *canvas.canvas.set()*.

18.1 Color Models and Named Colors

PyX provides four color classes corresponding to standard PostScript/PDF color models. All color component values are floating-point numbers in the range `[0.0, 1.0]`:

class *color.gray*(*g*)

Grayscale color with intensity *g* from 0.0 (black) to 1.0 (white). *grey* is provided as an alias for *gray*.

gray()

rgb()

cmyk()

hsb()

Converts the color to a *gray*, *rgb*, *cmyk*, or *hsb* instance.

class *color.rgb*(*r=0.0*, *g=0.0*, *b=0.0*)

RGB color with red (*r*), green (*g*), and blue (*b*) components in `[0.0, 1.0]`. Provides conversion methods *gray()*, *rgb()*, *cmyk()*, and *hsb()*, as well as *rgbhex()* (returning a "#RRGGBB" hex string).

class *color.cmyk*(*c=0.0*, *m=0.0*, *y=0.0*, *k=0.0*)

CMYK color with cyan (*c*), magenta (*m*), yellow (*y*), and black/key (*k*) ink components in `[0.0, 1.0]`. Provides conversion methods *gray()*, *rgb()*, *cmyk()*, and *hsb()*.

class *color.hsb*(*h=0.0*, *s=0.0*, *b=0.0*)

HSB (hue, saturation, brightness) color with *h*, *s*, and *b* in `[0.0, 1.0]`. Provides conversion methods *gray()*, *rgb()*, *cmyk()*, and *hsb()*.

In addition to constructing colors numerically, PyX predefines a rich palette of named colors as attributes on *gray*, *rgb*, and *cmyk* (such as *color.gray.black*, *color.rgb.red*, *color.cmyk.CornflowerBlue*). All predefined named colors are illustrated in appendix *Appendix: Named colors*.

18.1.1 Color Conversion Configuration

When converting between RGB, CMYK, and Grayscale via `c.rgb()`, `c.cmyk()`, or `c.gray()`, PyX uses configurable transfer functions that can be adjusted via `color.set()`:

18.1.2 Example

```
from pyx import *

c = canvas.canvas()

c.fill(path.rect(0, 0, 7, 3), [color.gray(0.8)])
c.fill(path.rect(1, 1, 1, 1), [color.rgb.red])
c.fill(path.rect(3, 1, 1, 1), [color.rgb.green])
c.fill(path.rect(5, 1, 1, 1), [color.rgb.blue])

c.writeEPSfile("color")
```

The resulting figure is shown in figure *Filling rectangles with grayscale and predefined RGB colors.*

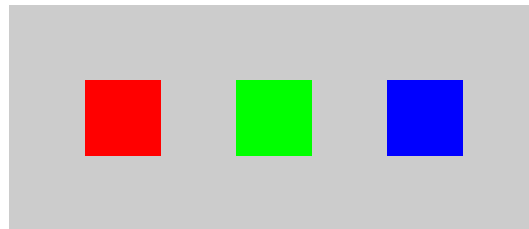


Fig. 1: Filling rectangles with grayscale and predefined RGB colors.

18.2 Color Gradients

The *gradient* hierarchy represents continuous transitions across a color space parameterized by a floating-point value in $[0, 1]$. Gradients serve two roles in PyX:

1. **Continuous color scales** (used by `deco.colorgradient`, `graph.style.rect`, `graph.style.surface`, and `graph.style.density`) via `gradient.getcolor()`.
2. **Changeable attributes** (`attr.changeattr`) that automatically step through N evenly spaced colors across a multi-dataset plot or `graph.style.bar` chart via `gradient.select()`.

All predefined gradients are listed and illustrated in appendix *Appendix: Named gradients*. Note that all predefined non-gray gradients are defined in the RGB color space except for `gradient.Rainbow`, `gradient.ReverseRainbow`, `gradient.Hue`, and `gradient.ReverseHue`, which are defined in HSB space; pre-converted RGB and CMYK versions of those four are provided as `rgbgradient.Rainbow`, `cmykgradient.Rainbow`, etc.

class color.gradient

Base class for all color gradients.

getcolor(parameter)

Returns the *color* instance corresponding to *parameter* in $[0.0, 1.0]$.

select(index, n_indices)

Implements the `attr.changeattr` interface: returns the *index*-th color out of *n_indices* evenly spaced samples along the gradient (evaluating `getcolor(index / (n_indices - 1))` when *n_indices* > 1, or `getcolor(0)` when *n_indices* == 1).

```
class color.functiongradient_gray(f_gray)
```

```
class color.functiongradient_rgb(f_r, f_g, f_b)
```

```
class color.functiongradient_cmyk(f_c, f_m, f_y, f_k)
```

```
class color.functiongradient_hsb(f_h, f_s, f_b)
```

Constructs a custom gradient in the respective color space where each component callable `f_x`(*x*) maps a parameter $x \in [0, 1]$ to a component value in $[0, 1]$.

```
color.lineargradient_gray(mincolor, maxcolor)
```

```
color.lineargradient_rgb(mincolor, maxcolor)
```

```
color.lineargradient_cmyk(mincolor, maxcolor)
```

```
color.lineargradient_hsb(mincolor, maxcolor)
```

Factory functions that return a `functiongradient_*` instance interpolating linearly between *mincolor* (at 0.0) and *maxcolor* (at 1.0) in the respective color space.

```
class color.rgbgradient(gradient)
```

```
class color.cmykgradient(gradient)
```

Wraps an existing *gradient* (such as an HSB gradient) and converts all returned colors to *rgb* or *cmyk*. This is particularly useful for `graph.style.surface` and `graph.style.density` plots in PostScript and PDF, which require shading meshes and bitmaps to use Gray, RGB, or CMYK color spaces rather than HSB.

18.3 Transparency

```
class color.transparency(value)
```

Makes stroking and filling operations partially transparent, where *value* is in the range 0.0 (completely opaque) to 1.0 (completely transparent).

Note

Transparency is natively supported in PDF and SVG output. Standard PostScript does not support alpha transparency; when writing PostScript or EPS, PyX emits a Ghostscript-specific transparency extension (supported by `ps2pdf`) and logs a warning.

MODULE *PATTERN*

The *pattern* module provides tiling patterns (*pattern*) that can be used as fill styles (and, in PostScript and SVG, stroke styles), along with predefined parallel-line (*hatched*) and grid (*crosshatched*) hatching patterns.

19.1 Class *pattern*

A *pattern* is a specialized *canvas.canvas* that also acts as a *style.fillstyle*. To define a custom tiling pattern, instantiate *pattern*, draw into it using standard canvas methods (*stroke()*, *fill()*, *text()*, etc.), and then pass the pattern instance as an attribute when filling a path:

```
from pyx import *

p = pattern.pattern()
p.fill(path.circle(0, 0, 0.1), [color.rgb.red])

c = canvas.canvas()
c.draw(path.rect(0, 0, 4, 2), [deco.filled([p]), deco.stroked()])
c.writePDFfile("polkadots")
```

class *pattern.pattern*(*painttype=1*, *tilingtype=1*, *xstep=None*, *ystep=None*, *bbox=None*, *trafo=None*,
 bboxenlarge=None, ***kwargs*)

Creates a new tiling pattern canvas. The constructor accepts the following arguments:

- *painttype*: 1 (default) for a colored pattern whose colors are specified when drawing into the pattern canvas, or 2 for an uncolored pattern (supported in PostScript and PDF) whose color is determined at the time the pattern is used.
- *tilingtype*: Controls how PostScript and PDF renderers adjust pattern spacing relative to device pixels:
 - 1 (default): Constant spacing between pattern cells (spaced by a multiple of a device pixel).
 - 2: Undistorted pattern cell (spacing between cells may vary by up to one device pixel).
 - 3: Constant spacing with faster tiling (allows slight cell distortion for faster rendering).
- *xstep*: Horizontal repetition step between pattern cells (a PyX length or number in user units), or *None* (default) to use the width of the pattern's bounding box.
- *ystep*: Vertical repetition step between pattern cells, or *None* (default) to use the height of the pattern's bounding box.
- *bbox*: Explicit *bbox.bbox* for the pattern cell, or *None* (default) to compute the bounding box automatically from the drawn items.
- *trafo*: Optional *trafo.trafo* applied to the pattern cell (for example, to rotate the tiling grid or shift its phase).

- *bboxenlarge*: Optional length by which an automatically computed bounding box is enlarged on all sides (default None).

19.2 Predefined Hatching Patterns

For common technical drawings and bar charts, *pattern* provides ready-to-use single-direction (*hatched*) and orthogonal two-direction (*crosshatched*) line patterns:

class *pattern.hatched*(*dist*, *angle*, *strokestyles*=[])

Creates a parallel-line hatching pattern with line spacing *dist* (a PyX length), rotation *angle* in degrees, and optional stroke attributes *strokestyles* (merged with [style.linewidth.THIN]). Calling an existing *hatched* instance *h*(*dist*=..., *angle*=..., *strokestyles*=...) returns a modified copy.

Predefined instances are provided for four standard angles (0, 45, 90, and 135 degrees), each with 13 spacing scales ranging from SMALL to LARGE (scaled by powers of $\sqrt{2}$ around a base spacing of $0.1 * \text{unit.v_cm}$):

- *hatched0* (vertical lines, *angle*=0): *hatched0*.SMALL, *hatched0*.SMALL, *hatched0*.SMALL, *hatched0*.SMALL, *hatched0*.Small, *hatched0*.Small, *hatched0*.small, *hatched0*.normal, *hatched0*.large, *hatched0*.Large, *hatched0*.Large, *hatched0*.Large, *hatched0*.LARGE, *hatched0*.LARGE
- *hatched45* (diagonal lines at 45°): *hatched45*.SMALL ... *hatched45*.LARGE
- *hatched90* (horizontal lines, *angle*=90): *hatched90*.SMALL ... *hatched90*.LARGE
- *hatched135* (diagonal lines at 135°): *hatched135*.SMALL ... *hatched135*.LARGE

class *pattern.crosshatched*(*dist*, *angle*, *strokestyles*=[])

Creates a two-directional orthogonal cross-hatching pattern with grid spacing *dist*, rotation *angle* in degrees, and optional *strokestyles* (merged with [style.linewidth.THIN]).

Predefined instances with the same SMALL ... normal ... LARGE size attributes are available for two orientations:

- *crosshatched0* (horizontal and vertical grid, *angle*=0)
- *crosshatched45* (diagonal grid at 45° and 135°, *angle*=45)

MODULE UNIT

The `unit` module provides classes and functions for specifying, scaling, and converting lengths in PyX. Every length in PyX combines a numerical value, a measurement unit ("m", "cm", "mm", "inch", or "pt"), and a combination of five semantic length categories:

"t" (true length)

Unscaled physical length. True lengths ignore all user scaling factors and are returned by PyX measurement methods such as `path.path.arclength()` or `bbox.bbox.width()`.

"u" (user length)

The default category for geometric coordinates, positions, and path dimensions. Plain numbers passed to non-`_pt` PyX classes and methods are automatically interpreted as user lengths in the current default unit.

"v" (visual length)

Used for the size of visual ornaments such as arrowheads, graph symbols, and axis tick marks. Scaling visual lengths changes the size of markers and labels without altering the overall geometry of a diagram.

"w" (width length)

Used for line widths (`style.linewidth`). Scaling width lengths lets you thicken or thin all strokes globally (for example, when preparing a figure for a presentation or high-DPI print).

"x" (TeX length)

Used to scale all TeX/LaTeX and `text.UnicodeEngine` text output.

Because a `length` instance stores all five components independently, you can freely add lengths of different types—such as `1 * unit.u_cm + 2 * unit.v_mm`—and PyX will resolve the combined length using the active scaling factors when drawing.

20.1 Configuring Units and Scaling Factors

Unless specified otherwise, plain numbers in PyX are interpreted in terms of the default unit, which is "cm" (1 centimeter). You can change the default unit and the scaling factors for the four scalable length categories using `unit.set()`:

```
unit.set(uscale=None, vscale=None, wscale=None, xscale=None, defaultunit=None)
```

Updates the scaling factors for user (*uscale*), visual (*vscale*), width (*wscale*), and TeX (*xscale*) lengths, or changes *defaultunit* (which must be one of "m", "cm", "mm", "inch", or "pt"). Any argument left as `None` retains its current setting.

For example, to double all line widths and increase text size by 20% for a slide deck while keeping the figure dimensions unchanged:

```
from pyx import *
unit.set(wscale=2.0, xscale=1.2)
```

Or to work in inches instead of centimeters throughout your script:

```
unit.set(defaultunit="inch")
```

20.2 Class *length*

`class unit.length(f=0, type='u', unit=None)`

Constructs a length with numerical factor *f*, length category *type* (one of "u", "v", "w", "x", or "t"; default "u"), and measurement *unit* (one of "m", "cm", "mm", "inch", or "pt"; defaults to the active *defaultunit*).

length instances support full arithmetic and comparison operations:

- **Addition and subtraction** ($11 + 12$, $11 - 12$): Sums or subtracts another *length* component-wise. If a plain number is added or subtracted, it is first converted to a user length in the current default unit.
- **Multiplication and division by a scalar** ($c * 1$, $1 / c$): Scales all five components of 1 by the number *c*, returning a new *length*.
- **Division of two lengths** ($11 / 12$): Converts both lengths to meters using the current scaling factors and returns their dimensionless float ratio.
- **Comparisons** ($11 == 12$, $11 < 12$, etc.): Evaluates both lengths in meters using the current scaling factors and compares the resulting values.

20.3 Predefined Length Constants

Rather than instantiating *length* directly, it is usually most convenient to multiply a number by one of the predefined unit constants in *unit*—for example, $5 * \text{unit.w_pt}$ or $1.5 * \text{unit.v_cm}$:

Constant	Type	Unit
m	user	meters
cm	user	centimeters
mm	user	millimeters
inch	user	inches
pt	user	points (1/72")
t_m	true	meters
t_cm	true	centimeters
t_mm	true	millimeters
t_inch	true	inches
t_pt	true	points (1/72")
u_m	user	meters
u_cm	user	centimeters
u_mm	user	millimeters
u_inch	user	inches
u_pt	user	points (1/72")
v_m	visual	meters
v_cm	visual	centimeters
v_mm	visual	millimeters
v_inch	visual	inches
v_pt	visual	points (1/72")
w_m	width	meters
w_cm	width	centimeters
w_mm	width	millimeters

continues on next page

Table 1 – continued from previous page

Constant	Type	Unit
<code>w_inch</code>	width	inches
<code>w_pt</code>	width	points (1/72")
<code>x_m</code>	TeX	meters
<code>x_cm</code>	TeX	centimeters
<code>x_mm</code>	TeX	millimeters
<code>x_inch</code>	TeX	inches
<code>x_pt</code>	TeX	points (1/72")

20.4 Conversion Functions

To evaluate a *length* (or a plain number, which is treated as a user length in the current default unit) as a `float` in a specific physical unit taking the current scaling factors into account, use the following conversion functions:

Function	Return value (float)
<code>tom(1)()</code>	Value of 1 in meters.
<code>tocm(1)()</code>	Value of 1 in centimeters.
<code>tomm(1)()</code>	Value of 1 in millimeters.
<code>toinch(1)()</code>	Value of 1 in inches.
<code>topt(1)()</code>	Value of 1 in PostScript points (1/72 inch).

Note

PyX internals use `topt()` extensively at the boundary between user-facing methods and low-level `_pt` classes.

MODULE *TRAFO*: LINEAR TRANSFORMATIONS

The *trafo* module provides affine (linear plus translation) coordinate transformations. Because *trafo* inherits from `baseclasses.deformer`, transformation instances can be used seamlessly in several ways across PyX:

- Passed in the attribute list of `canvas.canvas.stroke()`, `canvas.canvas.fill()`, `canvas.canvas.draw()`, `canvas.canvas.insert()`, or `canvas.canvas.text()` to transform individual graphical objects.
- Passed to `canvas.canvas([trafo.scale(2)])` or `c.set([trafo.rotate(45)])` to transform the coordinate system of an entire canvas.
- Applied directly to a path via `p.transformed(t)` or `t.deform(p)`, or to a bounding box via `b.transformed(t)`.

21.1 Mathematical Definition and Composition

An affine transformation maps a 2D point vector $\vec{x} = (x, y)^T$ to

$$\vec{x}' = A\vec{x} + \vec{b},$$

where A is a non-singular 2×2 matrix ($\det A \neq 0$) and $\vec{b}$ is a 2D translation vector.

Transformations can be composed in two ways:

1. **Multiplication** (`t1 * t2`): Multiplying two transformations corresponds to matrix composition—`t1 * t2` applies `t2` *first*, followed by `t1`:

$$A = A_1 A_2, \quad \vec{b} = A_1 \vec{b}_2 + \vec{b}_1.$$

2. **Chained methods** (`t.translated(...)`, `t.rotated(...)`, etc.): Every *trafo* instance provides methods that return a new transformation obtained by applying the new operation *after* `t`. Thus `t.rotated(45)` is equivalent to `trafo.rotate(45) * t`:

```
# Translate by (2, 1) first, then rotate by 30 degrees around the origin:
t = trafo.translate(2, 1).rotated(30)
# Equivalent to:
t = trafo.rotate(30) * trafo.translate(2, 1)
```

21.2 Class *trafo*

class `trafo.trafo`(*matrix*=(1, 0), (0, 1)), *vector*=(0, 0))

Constructs an affine transformation with 2×2 transformation *matrix* ((a11, a12), (a21, a22)) and translation *vector* (bx, by) (where bx and by are PyX lengths or numbers in user units).

For internal point-based coordinates, `trafo_pt` takes *vector* directly in PostScript points. The module-level constant `trafo.identity` provides a preconstructed identity transformation.

apply(*x*, *y*)

Applies the transformation to the point (*x*, *y*) (in user units or PyX lengths) and returns the transformed point as a tuple of two true lengths (*tx*, *ty*).

apply_pt(*x_pt*, *y_pt*)

Applies the transformation to the point (*x_pt*, *y_pt*) in PostScript points and returns (*tx_pt*, *ty_pt*) in points.

deform(*path*)

Returns a transformed copy of *path* (equivalent to `path.transformed(self)`).

inverse()

Returns the inverse transformation A^{-1} with translation vector $-A^{-1}\vec{b}$.

mirrored(*angle*)

Returns `self` followed by a reflection across the line through the origin at *angle* degrees counter-clockwise from the positive *x*-axis.

rotated(*angle*, *x*=None, *y*=None)

Returns `self` followed by a counter-clockwise rotation by *angle* degrees around the center (*x*, *y*) (or (0, 0) if omitted). Also available in point coordinates as `rotated_pt()`.

scaled(*sx*, *sy*=None, *x*=None, *y*=None)

Returns `self` followed by scaling by *sx* in the *x*-direction and *sy* in the *y*-direction (if *sy* is None, uniform scaling *sy* = *sx* is used) relative to the center (*x*, *y*) (or (0, 0) if omitted). Also available in point coordinates as `scaled_pt()`.

slanted(*a*, *angle*=0, *x*=None, *y*=None)

Returns `self` followed by a shear (slant) of factor *a* along the direction *angle* (in degrees) relative to (*x*, *y*) (or (0, 0) if omitted). Also available in point coordinates as `slanted_pt()`.

translated(*x*, *y*)

Returns `self` followed by a translation by (*x*, *y*). Also available in point coordinates as `translated_pt()`.

21.3 Standard Transformation Subclasses

For each transformation operation, `trafo` provides a dedicated subclass of `trafo` (along with a corresponding `_pt` class taking coordinates in PostScript points):

class `trafo.mirror`(*angle*=0)

Reflection across the line through the origin (0, 0) at *angle* degrees from the positive *x*-axis.

class `trafo.rotate`(*angle*, *x*=None, *y*=None)

class `trafo.rotate_pt`(*angle*, *x*=None, *y*=None)

Counter-clockwise rotation by *angle* degrees around the point (*x*, *y*), or around (0, 0) if *x* and *y* are omitted.

class `trafo.scale`(*sx*, *sy*=None, *x*=None, *y*=None)

class trafo.**scale_pt**(*sx*, *sy=None*, *x=None*, *y=None*)

Scaling by factor *sx* along the *x*-axis and *sy* along the *y*-axis (defaults to *sy* = *sx* for uniform scaling) centered at (*x*, *y*), or at (0, 0) if *x* and *y* are omitted.

class trafo.**slant**(*a*, *angle=0*, *x=None*, *y=None*)

class trafo.**slant_pt**(*a*, *angle=0*, *x=None*, *y=None*)

Shear transformation by factor *a* along the direction *angle* (in degrees) centered at (*x*, *y*), or at (0, 0) if *x* and *y* are omitted. For *angle=0*, this maps $(x, y) \mapsto (x + ay, y)$.

class trafo.**translate**(*x*, *y*)

class trafo.**translate_pt**(*x*, *y*)

Translation by the vector (*x*, *y*).

APPENDIX: NAMED COLORS

The following chart lists the predefined grayscale, RGB, and CMYK colors available in the *color* module (see chapter *Module color*).

×  gray.black	×  cmyk.RubineRed	×  cmyk.Cerulean
×  gray.white	×  cmyk.WildStrawberry	×  cmyk.Cyan
	×  cmyk.Salmon	×  cmyk.ProcessBlue
×  rgb.red	×  cmyk.CarnationPink	×  cmyk.SkyBlue
×  rgb.green	×  cmyk.Magenta	×  cmyk.Turquoise
×  rgb.blue	×  cmyk.VioletRed	×  cmyk.TealBlue
×  rgb.white	×  cmyk.Rhodamine	×  cmyk.Aquamarine
×  rgb.black	×  cmyk.Mulberry	×  cmyk.BlueGreen
	×  cmyk.RedViolet	×  cmyk.Emerald
×  cmyk.GreenYellow	×  cmyk.Fuchsia	×  cmyk.JungleGreen
×  cmyk.Yellow	×  cmyk.Lavender	×  cmyk.SeaGreen
×  cmyk.Goldenrod	×  cmyk.Thistle	×  cmyk.Green
×  cmyk.Dandelion	×  cmyk.Orchid	×  cmyk.ForestGreen
×  cmyk.Apricot	×  cmyk.DarkOrchid	×  cmyk.PineGreen
×  cmyk.Peach	×  cmyk.Purple	×  cmyk.LimeGreen
×  cmyk.Melon	×  cmyk.Plum	×  cmyk.YellowGreen
×  cmyk.YellowOrange	×  cmyk.Violet	×  cmyk.SpringGreen
×  cmyk.Orange	×  cmyk.RoyalPurple	×  cmyk.OliveGreen
×  cmyk.BurntOrange	×  cmyk.BlueViolet	×  cmyk.RawSienna
×  cmyk.Bittersweet	×  cmyk.Periwinkle	×  cmyk.Sepia
×  cmyk.RedOrange	×  cmyk.CadetBlue	×  cmyk.Brown
×  cmyk.Mahogany	×  cmyk.CornflowerBlue	×  cmyk.Tan
×  cmyk.Maroon	×  cmyk.MidnightBlue	×  cmyk.Gray
×  cmyk.BrickRed	×  cmyk.NavyBlue	×  cmyk.Black
×  cmyk.Red	×  cmyk.RoyalBlue	×  cmyk.White
×  cmyk.OrangeRed	×  cmyk.Blue	

Fig. 1: Predefined named colors in the *color* module.

APPENDIX: NAMED GRADIENTS

The following chart illustrates the predefined color gradients available on `color.gradient`, `color.rgbgradient`, and `color.cmykgradient` (see chapter *Module color*).

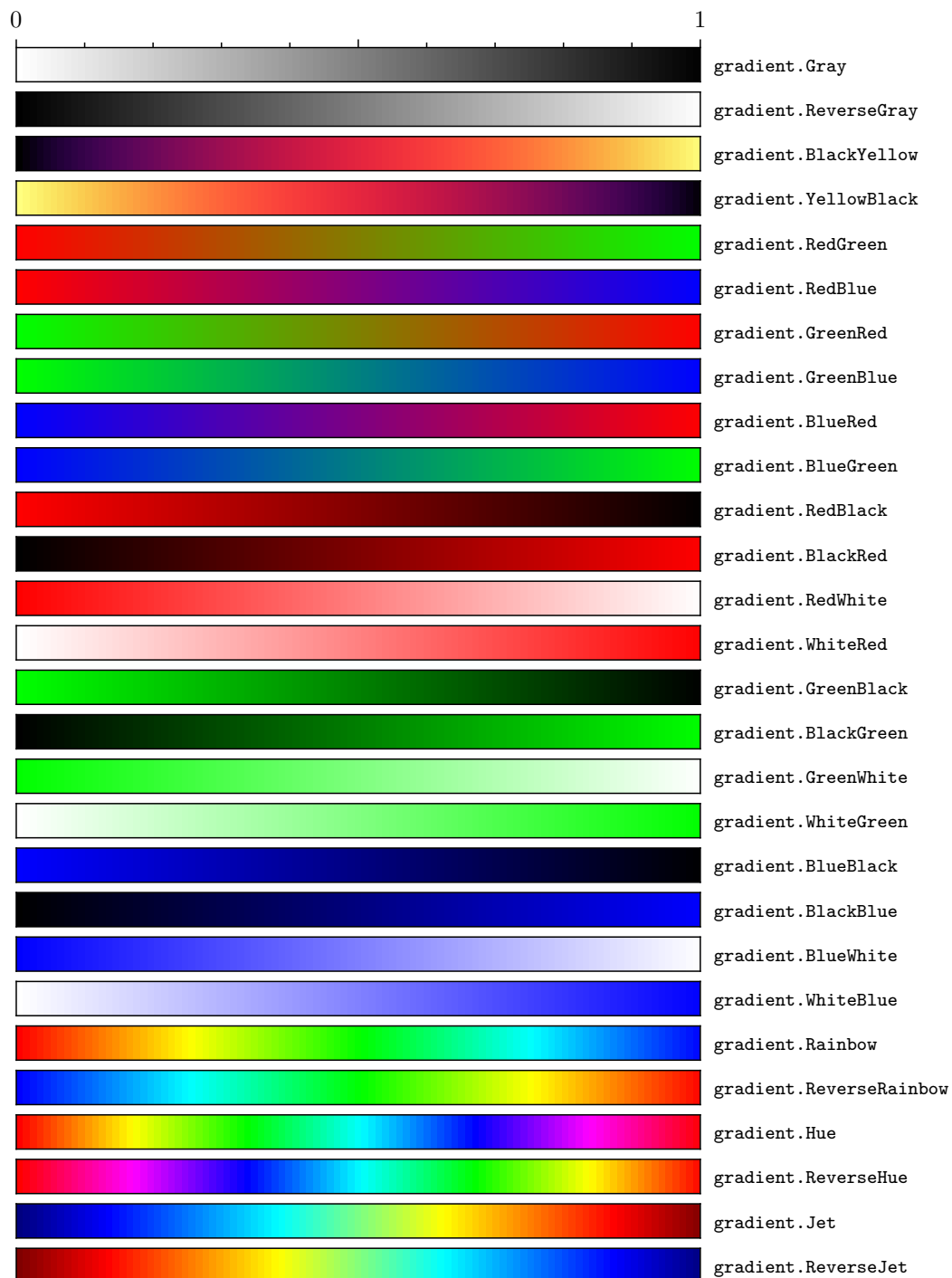


Fig. 1: Predefined color gradients in the *color* module.

APPENDIX: PATH STYLES IN *STYLE* MODULE

The following figure illustrates the predefined line caps (*linecap*), line joins (*linejoin*), dash patterns (*linestyle*), and line widths (*linewidth*) provided by the *style* module (see section *Styles: style* for detailed documentation).

	<code>linecap.butt</code> (default)		<code>miterlimit.less than 180deg</code>
	<code>linecap.round</code>		<code>miterlimit.less than 90deg</code>
	<code>linecap.square</code>		<code>miterlimit.less than 60deg</code>
			<code>miterlimit.less than 45deg</code>
	<code>linejoin.miter</code> (default)		<code>miterlimit.less than 11deg</code> (default)
	<code>linejoin.round</code>		
	<code>linejoin.bevel</code>		<code>dash((1, 1, 2, 2, 3, 3), 0)</code>
			<code>dash((1, 1, 2, 2, 3, 3), 1)</code>
	<code>linestyle.solid</code> (default)		<code>dash((1, 2, 3), 2)</code>
	<code>linestyle.dashed</code>		<code>dash((1, 2, 3), 3)</code>
	<code>linestyle.dotted</code>		<code>dash((1, 2, 3), 4)</code>
	<code>linestyle.dashdotted</code>		<code>dash((1, 2, 3), rellengths=1)</code>
	<code>linewidth.THIN</code>		
	<code>linewidth.THIn</code>		
	<code>linewidth.ThIn</code>		
	<code>linewidth.Thin</code>		
	<code>linewidth.thin</code>		
	<code>linewidth.normal</code> (default)		
	<code>linewidth.thick</code>		
	<code>linewidth.Thick</code>		
	<code>linewidth.THick</code>		
	<code>linewidth.THICK</code>		
	<code>linewidth.THICK</code>		
	<code>linewidth.THICK</code>		

Fig. 1: Predefined stroke styles in the `style` module.

APPENDIX: ARROWS IN *DECO* MODULE

The following figure illustrates the predefined ending (*deco.earrow*) and beginning (*deco.barrow*) arrow decorators and their customization in the *deco* module (see section *Decorators: deco* for detailed documentation).

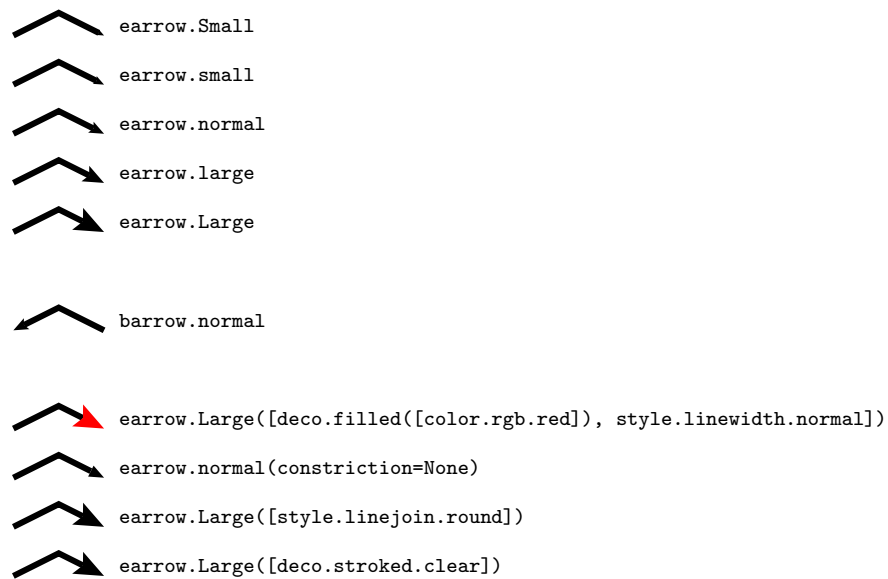


Fig. 1: Predefined arrow decorators in the *deco* module.

MODULE *UTILS*: HELPER UTILITIES

The *utils* module provides internal helper functions used by PyX's graph and axis classes to support hierarchical keyword argument forwarding (such as passing `x_painter_gridattrs=...` directly to `graph.graphxy`) and cloning configurable component instances via `__call__`.

`utils.kwsplit(kwargs, parts=None, allow_only_split=False)`

Splits a dictionary *kwargs* of keyword arguments at the first underscore "_" into nested dictionaries.

- Any key of the form "<part>_<subkey>" (where <part> is in *parts*, or for any <part> if *parts*=None) is placed into `split[part][subkey]`.
- All remaining keys are placed into `rest[key]` (unless `allow_only_split=True`, in which case unsplit keys raise a `ValueError`).
- Returns the tuple (`split`, `rest`).

`utils.merge_members_kwargs(obj, kwargs_updates, member_names)`

Builds a merged keyword dictionary starting from the attributes listed in *member_names* on *obj* and sequentially applying each dictionary in the sequence *kwargs_updates*. Used by axis, painter, and texter `__call__` methods to return a modified clone of an existing component instance.

PYTHON MODULE INDEX

a

`attr`, 14

b

`bbox`, 105

`bitmap`, 101

`box`, 87

c

`canvas`, 39

`color`, 107

`config`, 6

`connector`, 91

d

`deco`, 18

`deformer`, 35

`document`, 43

e

`epsfile`, 95

g

`graph`, 63

`graph.axis`, 77

`graph.axis.axis`, 78

`graph.axis.painter`, 84

`graph.axis.parter`, 82

`graph.axis.positioner`, 86

`graph.axis.rater`, 86

`graph.axis.texter`, 83

`graph.axis.tick`, 81

`graph.axis.timeaxis`, 81

`graph.data`, 70

`graph.graph`, 66

`graph.key`, 76

`graph.style`, 72

m

`metapost.path`, 31

n

`normpath`, 27

p

`path`, 23

`pattern`, 111

`pdffile`, 97

s

`style`, 16

`svgfile`, 99

t

`text`, 47

`trafo`, 117

u

`unit`, 113

`utils`, 127

Symbols

`_positioner` (class in `graph.axis.positioner`), 86

A

`anchoredaxis` (class in `graph.axis.axis`), 80
`anchoredpathaxis` (class in `graph.axis.axis`), 80
`append()` (`document.document` method), 44
`append()` (`normpath.normsubpath` method), 29
`append()` (`path.path` method), 26
`apply()` (`trafo.trafo` method), 118
`apply_pt()` (`trafo.trafo` method), 118
`arc` (class in `connector`), 92
`arc` (class in `path`), 24
`arc_pt` (class in `path`), 24
`arclen()` (`normpath.normpathparam` method), 28
`arclen()` (`path.path` method), 26
`arclen_pt()` (`normpath.normpathparam` method), 28
`arclen_pt()` (`path.path` method), 26
`arclentoparam()` (`path.path` method), 26
`arclentoparam_pt()` (`path.path` method), 26
`arcn` (class in `path`), 25
`arcn_pt` (class in `path`), 25
`arct` (class in `path`), 25
`arct_pt` (class in `path`), 25
`arrow` (class in `deco`), 18
`arrow` (class in `graph.style`), 74
`at()` (`path.path` method), 26
`at_pt()` (`path.path` method), 26
`atbegin()` (`path.path` method), 26
`atbegin_pt()` (`path.path` method), 26
`atend()` (`path.path` method), 26
`atend_pt()` (`path.path` method), 26
`attr`
 module, 14
`attr` (class in `attr`), 14
`autolin` (class in `graph.axis.parter`), 82
`autolinear` (class in `graph.axis.parter`), 82
`autolog` (class in `graph.axis.parter`), 82
`autologarithmic` (class in `graph.axis.parter`), 82
`autosizedlinear` (class in `graph.axis.axis`), 80
`axes` (`graph.graph.graphxy` attribute), 67
`axisatv()` (`graph.graph.graphxy` method), 68

B

`bar` (class in `graph.axis.axis`), 79
`bar` (class in `graph.axis.painter`), 85
`bar` (class in `graph.style`), 75
`barpos` (class in `graph.style`), 75
`barrow` (in module `deco`), 19
`baseline` (`text.valign` attribute), 57
`basepath()` (`graph.axis.axis.anchoredaxis` method), 80
`bbox`
 module, 103
`bbox` (class in `bbox`), 105
`bbox()` (`box.polygon` method), 88
`bbox()` (`canvas.canvas` method), 40
`bbox()` (`path.path` method), 26
`bbox_pt` (class in `bbox`), 105
`begin()` (`path.path` method), 26
`beginknot` (class in `metapost.path`), 32
`beginknot_pt` (class in `metapost.path`), 32
`bitmap`
 module, 100
`bitmap` (class in `bitmap`), 102
`bottom` (`text.valign` attribute), 57
`bottomzero` (`text.vshift` attribute), 57
`box`
 module, 86
`box_warning()` (`text.texmessage` static method), 55
`boxcenter` (`text.halign` attribute), 56
`BoxCrossError`, 88
`boxdistance()` (`box.polygon` method), 88
`boxdistance_pt()` (`box.polygon` method), 88
`boxleft` (`text.halign` attribute), 56
`boxright` (`text.halign` attribute), 56
`brace` (class in `deco`), 20

C

`canvas`
 module, 39
`canvas` (class in `canvas`), 40
`cdbfile` (class in `graph.data`), 71
`center` (`text.halign` attribute), 56
`changeattr` (class in `attr`), 15
`changebar` (class in `graph.style`), 75

`changelist` (class in *attr*), 15
`checkattrs()` (in module *attr*), 15
`circle` (class in *path*), 30
`circle_pt` (class in *path*), 30
`circlealign()` (box.polygon method), 88
`circlealign_pt()` (box.polygon method), 88
`circlealignequal()` (in module *box*), 88
`circlealignequal_pt()` (in module *box*), 88
`circlealignvector()` (box.polygon method), 87
`circlealignvector_pt()` (box.polygon method), 87
`clear` (in module *attr*), 15
`clearclass` (class in *attr*), 15
`clip` (class in *canvas*), 42
`close()` (normpath.normsubpath method), 29
`closed` (normpath.normsubpath attribute), 29
`closepath` (class in *path*), 24
`cmyk` (class in *color*), 107
`cmyk()` (color.gray method), 107
`cmykgradient` (class in *color*), 109
`color`
 module, 106
`colorgradient` (class in *deco*), 20
`conffile` (class in *graph.data*), 71
`config`
 module, 6
`connector`
 module, 89
`controlcurve` (class in *metapost.path*), 33
`controlcurve_pt` (class in *metapost.path*), 33
`copy()` (normpath.normpath method), 29
`cornersmoothed` (class in *deformer*), 35
`crosshatched` (class in *pattern*), 112
`cube` (class in *graph.axis.rater*), 86
`curvature_pt()` (path.path method), 26
`curve` (class in *connector*), 92
`curve` (class in *metapost.path*), 33
`curve` (class in *path*), 30
`curve_pt` (class in *path*), 30
`curvedtext` (class in *deco*), 20
`curveto` (class in *path*), 25
`curveto_pt` (class in *path*), 25
`cycloid` (class in *deformer*), 37

D

`dash` (class in *style*), 16
`data` (class in *graph.data*), 71
`decimal` (class in *graph.axis.texter*), 83
`deco`
 module, 18
`default` (class in *graph.axis.texter*), 83
`default` (text.errordetail attribute), 54
`defaultcolumnpattern` (graph.data.file attribute), 71
`defaultcommentpattern` (graph.data.file attribute), 71
`defaultstringpattern` (graph.data.file attribute), 71

`defaulttextengine` (in module *text*), 53
`defaultvariants` (graph.axis.parter.autolin.autolinear attribute), 82
`deform()` (deformer.cornersmoothed method), 36
`deform()` (deformer.cycloid method), 37
`deform()` (deformer.parallel method), 36
`deform()` (trafo.trafo method), 118
`deformer`
 module, 35
`density` (class in *graph.style*), 76
`depth` (text.texttextbox_pt attribute), 52
`distance` (class in *graph.axis.rater*), 86
`doaxes()` (graph.graph.graphxy method), 67
`dobackground()` (graph.graph.graphxy method), 67
`document`
 module, 43
`document` (class in *document*), 44
`dokey()` (graph.graph.graphxy method), 68
`dokeyitem()` (graph.graph.graphxy method), 68
`dolayout()` (graph.graph.graphxy method), 67
`doplot()` (graph.graph.graphxy method), 68
`doplotitem()` (graph.graph.graphxy method), 68
`doranges()` (graph.graph.graphxy method), 67
`dostyles()` (graph.graph.graphxy method), 68
`draw()` (canvas.canvas method), 40

E

`earrow` (in module *deco*), 19
`ellipse` (class in *path*), 30
`ellipse_pt` (class in *path*), 30
`empty` (class in *bbox*), 105
`end()` (path.path method), 26
`end()` (text.texmessage static method), 54
`endknot` (class in *metapost.path*), 32
`endknot_pt` (class in *metapost.path*), 32
`epsfile`
 module, 93
`epsfile` (class in *epsfile*), 95
`epsilon` (normpath.normsubpath attribute), 29
`errorbar` (class in *graph.style*), 74
`errordetail` (class in *text*), 54
`escapestring()` (in module *text*), 53
`exclusiveattr` (class in *attr*), 15
`extend()` (normpath.normsubpath method), 29
`extend()` (path.path method), 26
`extent()` (box.polygon method), 88
`extent_pt()` (box.polygon method), 88

F

`file` (class in *graph.data*), 70
`fill()` (canvas.canvas method), 40
`filled` (in module *deco*), 18
`fillrule` (class in *style*), 17
`finish()` (graph.graph.graphxy method), 68

flushcenter (*text.halign attribute*), 56
 flushleft (*text.halign attribute*), 56
 flushright (*text.halign attribute*), 56
 font_warning() (*text.texmessage static method*), 55
 footnotesize (*text.size attribute*), 58
 full (*text.errordetail attribute*), 54
 function (*class in graph.data*), 71
 functiongradient_cmyk (*class in color*), 109
 functiongradient_gray (*class in color*), 109
 functiongradient_hsb (*class in color*), 109
 functiongradient_rgb (*class in color*), 109

G

get() (*in module config*), 6
 getattrs() (*in module attr*), 15
 getboolean() (*in module config*), 6
 getcolor() (*color.gradient method*), 108
 getfloat() (*in module config*), 6
 getint() (*in module config*), 6
 getlist() (*in module config*), 6
 gradient (*class in color*), 108
 graph
 module, 61
 graph.axis
 module, 76
 graph.axis.axis
 module, 78
 graph.axis.painter
 module, 84
 graph.axis.parter
 module, 81
 graph.axis.positioner
 module, 86
 graph.axis.rater
 module, 85
 graph.axis.texter
 module, 82
 graph.axis.tick
 module, 81
 graph.axis.timeaxis
 module, 81
 graph.data
 module, 69
 graph.graph
 module, 66
 graph.key
 module, 76
 graph.style
 module, 72
 graphx (*class in graph.graph*), 68
 graphxy (*class in graph.graph*), 66
 graphxyz (*class in graph.graph*), 69
 graphxyz.central (*class in graph.graph*), 69
 graphxyz.parallel (*class in graph.graph*), 69

gray (*class in color*), 107
 gray() (*color.gray method*), 107
 grid (*class in graph.style*), 75
 gridpath() (*graph.axis.axis.anchoredaxis method*), 80
 gridpos (*class in graph.style*), 75

H

halign (*class in text*), 56
 hatched (*class in pattern*), 112
 height (*text.texttextbox_pt attribute*), 52
 histogram (*class in graph.style*), 74
 hsb (*class in color*), 107
 hsb() (*color.gray method*), 107
 Huge (*text.size attribute*), 58
 huge (*text.size attribute*), 58

I

ignore() (*text.texmessage static method*), 55
 image (*class in bitmap*), 102
 impulses (*class in graph.style*), 74
 insert() (*canvas.canvas method*), 41
 intersect() (*path.path method*), 27
 inverse() (*trafo.trafo method*), 118
 items (*canvas.canvas attribute*), 40

J

join() (*path.path method*), 27
 joined() (*path.path method*), 27
 jpegimage (*class in bitmap*), 102

K

key (*class in graph.key*), 76
 knot (*class in metapost.path*), 32
 kwsplit() (*in module utils*), 129

L

LARGE (*text.size attribute*), 58
 Large (*text.size attribute*), 58
 large (*text.size attribute*), 58
 LatexEngine (*class in text*), 51
 layer() (*canvas.canvas method*), 41
 layers (*canvas.canvas attribute*), 40
 left (*text.halign attribute*), 56
 left (*text.texttextbox_pt attribute*), 52
 length (*class in unit*), 114
 lin (*class in graph.axis.axis*), 78
 lin (*class in graph.axis.parter*), 82
 lin (*class in graph.axis.rater*), 86
 line (*class in connector*), 92
 line (*class in graph.style*), 74
 line (*class in metapost.path*), 33
 line (*class in path*), 30
 line_pt (*class in path*), 30

`linealign()` (*box.polygon method*), 88
`linealign_pt()` (*box.polygon method*), 88
`linealignequal()` (*in module box*), 89
`linealignequal_pt()` (*in module box*), 89
`linealignvector()` (*box.polygon method*), 87
`linealignvector_pt()` (*box.polygon method*), 88
`linear` (*class in graph.axis.axis*), 78
`linear` (*class in graph.axis.parter*), 82
`linear` (*class in graph.axis.rater*), 86
`lineargradient_cmyk()` (*in module color*), 109
`lineargradient_gray()` (*in module color*), 109
`lineargradient_hsb()` (*in module color*), 109
`lineargradient_rgb()` (*in module color*), 109
`linecap` (*class in style*), 16
`linehatched` (*class in deco*), 20
`linejoin` (*class in style*), 16
`linestyle` (*class in style*), 16
`lineto` (*class in path*), 24
`lineto_pt` (*class in path*), 24
`linewidth` (*class in style*), 17
`linked` (*class in graph.axis.painter*), 85
`linkedaxis` (*class in graph.axis.axis*), 80
`linkedbar` (*class in graph.axis.painter*), 85
`linkedsplit` (*class in graph.axis.painter*), 85
`load()` (*text.texmessage static method*), 54
`load_def()` (*text.texmessage static method*), 55
`load_graphics()` (*text.texmessage static method*), 55
`log` (*class in graph.axis.axis*), 79
`log` (*class in graph.axis.parter*), 82
`log` (*class in graph.axis.rater*), 86
`logarithmic` (*class in graph.axis.axis*), 79
`logarithmic` (*class in graph.axis.parter*), 82
`logarithmic` (*class in graph.axis.rater*), 86

M

`marker()` (*text.texttextbox_pt method*), 52
`mathaxis` (*text.vshift attribute*), 58
`mathmode` (*in module text*), 58
`merge_members_kwargs()` (*in module utils*), 129
`mergeattrs()` (*in module attr*), 15
`metapost.path`
 module, 31
`middle` (*text.valign attribute*), 57
`middlezero` (*text.vshift attribute*), 58
`mirror` (*class in trafo*), 118
`mirrored()` (*trafo.trafo method*), 118
`miterlimit` (*class in style*), 16
`module`
 attr, 14
 bbox, 103
 bitmap, 100
 box, 86
 canvas, 39
 color, 106

config, 6
 connector, 89
 deco, 18
 deformer, 35
 document, 43
 epsfile, 93
 graph, 61
 graph.axis, 76
 graph.axis.axis, 78
 graph.axis.painter, 84
 graph.axis.parter, 81
 graph.axis.positioner, 86
 graph.axis.rater, 85
 graph.axis.texter, 82
 graph.axis.tick, 81
 graph.axis.timeaxis, 81
 graph.data, 69
 graph.graph, 66
 graph.key, 76
 graph.style, 72
 metapost.path, 31
 normpath, 27
 path, 23
 pattern, 109
 pdffile, 96
 style, 16
 svgfile, 98
 text, 45
 trafo, 115
 unit, 112
 utils, 127

`moveto` (*class in path*), 24
`moveto_pt` (*class in path*), 24
`multichangeattr` (*class in attr*), 15
`multicurveto_pt` (*class in path*), 25
`MultiEngine` (*class in text*), 51
`MultiEngineText` (*class in text*), 59
`multilineto_pt` (*class in path*), 24

N

`nestedbar` (*class in graph.axis.axis*), 79
`no_aux()` (*text.texmessage static method*), 54
`no_file()` (*text.texmessage static method*), 54
`no_nav()` (*text.texmessage static method*), 54
`nobbl_warning()` (*text.texmessage static method*), 56
`none` (*text.errordetail attribute*), 54
`normalsize` (*text.size attribute*), 58
`normcurve_pt` (*class in normpath*), 29
`normline_pt` (*class in normpath*), 29
`normpath`
 module, 27
`normpath` (*class in normpath*), 28
`normpath()` (*path.path method*), 27
`normpathparam` (*class in normpath*), 28

`normsubpath` (class in `normpath`), 29

`normsubpathitems` (`normpath.normsubpath` attribute), 29

O

`open()` (in module `config`), 6

`orthogonal` (`graph.axis.painter.rotatetext` attribute), 84

P

`package_warning()` (`text.texmessage` static method), 55

`page` (class in `document`), 43

`palette` (class in `bitmap`), 102

`paperformat` (class in `document`), 44

`parallel` (class in `deformer`), 36

`parallel` (`graph.axis.painter.rotatetext` attribute), 84

`paramfunction` (class in `graph.data`), 72

`paramtoarclen()` (`path.path` method), 26

`paramtoarclen_pt()` (`path.path` method), 26

`parbox` (class in `text`), 57

`path`

module, 23

`path` (class in `metapost.path`), 32

`path` (class in `path`), 25

`path()` (`box.polygon` method), 87

`path()` (`normpath.normpath` method), 29

`path()` (`path.path` method), 27

`pathaxis()` (in module `graph.axis.axis`), 80

`pathitems()` (`normpath.normsubpath` method), 29

`pattern`

module, 109

`pattern` (class in `pattern`), 111

`pattern()` (`text.texmessage` static method), 55

`pdffile`

module, 96

`pdffile` (class in `pdffile`), 97

`phantom` (in module `text`), 58

`pipeGS()` (in module `canvas`), 41

`plot()` (`graph.graph.graphxy` method), 67

`pointdistance()` (`box.polygon` method), 88

`pointdistance_pt()` (`box.polygon` method), 88

`points` (class in `graph.data`), 70

`polygon` (class in `box`), 87

`pos` (class in `graph.style`), 72

`pos()` (`graph.graph.graphxy` method), 68

`pos_pt()` (`graph.graph.graphxy` method), 68

`preamble` (in module `text`), 53

`preamble()` (`text.MultiEngine` method), 51

`preamble()` (`text.SingleEngine` method), 49

`preexp` (class in `graph.axis.parter`), 82

`pyxinfo()` (in module `pyx`), 61

R

`raggedcenter` (`text.halign` attribute), 56

`raggedleft` (`text.halign` attribute), 56

`raggedright` (`text.halign` attribute), 56

`range` (class in `graph.style`), 73

`rater` (class in `graph.axis.rater`), 86

`rational` (class in `graph.axis.texter`), 83

`rational` (class in `graph.axis.tick`), 81

`rcurveto` (class in `path`), 25

`rcurveto_pt` (class in `path`), 25

`rect` (class in `box`), 89

`rect` (class in `graph.style`), 75

`rect` (class in `path`), 30

`rect_pt` (class in `path`), 30

`refineattrs()` (in module `attr`), 15

`regular` (class in `graph.axis.painter`), 84

`reltransform()` (`box.polygon` method), 87

`rerun_warning()` (`text.texmessage` static method), 55

`reset` (in module `text`), 53

`reset()` (`text.MultiEngine` method), 51

`reversed()` (`path.path` method), 27

`rgb` (class in `color`), 107

`rgb()` (`color.gray` method), 107

`rgbgradient` (class in `color`), 109

`right` (`text.halign` attribute), 56

`right` (`text.texttextbox_pt` attribute), 52

`rlineto` (class in `path`), 24

`rlineto_pt` (class in `path`), 24

`rmoveto` (class in `path`), 24

`rmoveto_pt` (class in `path`), 24

`rotate` (class in `trafo`), 118

`rotate_pt` (class in `trafo`), 118

`rotated()` (`trafo.trafo` method), 118

`rotatetext` (class in `graph.axis.painter`), 84

`rotation()` (`path.path` method), 27

`rotation_pt()` (`path.path` method), 27

`roughknot` (class in `metapost.path`), 32

`roughknot_pt` (class in `metapost.path`), 32

S

`scale` (class in `trafo`), 118

`scale_pt` (class in `trafo`), 118

`scaled()` (`trafo.trafo` method), 118

`scriptsize` (`text.size` attribute), 58

`segments()` (`normpath.normsubpath` method), 29

`select()` (`color.gradient` method), 108

`selectattr()` (in module `attr`), 15

`selectattrs()` (in module `attr`), 15

`set()` (in module `metapost.path`), 32

`set()` (in module `normpath`), 28

`set()` (in module `text`), 53

`set()` (in module `unit`), 113

`setlinkedaxis()` (`graph.axis.axis.linkedaxis` method), 80

`settextengine()` (`canvas.canvas` method), 41

`shownormpath` (class in `deco`), 21

`SingleEngine` (class in `text`), 48

SingleLatexEngine (class in text), 50
SingleTexEngine (class in text), 50
size (class in text), 58
sizedlinear (class in graph.axis.axis), 80
slant (class in trafo), 119
slant_pt (class in trafo), 119
slanted() (trafo.trafo method), 118
small (text.size attribute), 58
smoothed (class in deformer), 35
smoothknot (class in metapost.path), 32
smoothknot_pt (class in metapost.path), 32
sortbeforeattr (class in attr), 15
sortbeforeexclusiveattr (class in attr), 15
split (class in graph.axis.axis), 79
split (class in graph.axis.painter), 85
split() (path.path method), 27
split_pt() (path.path method), 27
splitatvalue() (in module graph.data), 72
stackedbarpos (class in graph.style), 75
StackedText (class in text), 59
start() (text.texmessage static method), 54
startknot (class in metapost.path), 32
stroke() (canvas.canvas method), 40
stroked (in module deco), 18
style
 module, 16
surface (class in graph.style), 75
svgfile
 module, 98
svgfile (class in svgfile), 99
svgfile_pt (class in svgfile), 99
symbol (class in graph.style), 73

T

tangent() (path.path method), 27
tangent_pt() (path.path method), 27
tensioncurve (class in metapost.path), 33
TexEngine (class in text), 51
texmessage (class in text), 54
texmessages_begindoc_default
 (text.SingleLatexEngine attribute), 51
texmessages_docclass_default
 (text.SingleLatexEngine attribute), 51
texmessages_end_default (text.SingleEngine at-
 tribute), 49
texmessages_preamble_default (text.SingleEngine
 attribute), 49
texmessages_run_default (text.SingleEngine at-
 tribute), 49
texmessages_start_default (text.SingleEngine at-
 tribute), 49
TexResultError, 54
text
 module, 45

text (class in deco), 19
text (class in graph.style), 74
Text (class in text), 59
text (in module text), 53
text() (canvas.canvas method), 41
text() (text.MultiEngine method), 51
text() (text.SingleEngine method), 50
text() (text.UnicodeEngine method), 59
text_pt (in module text), 53
text_pt() (canvas.canvas method), 41
text_pt() (text.MultiEngine method), 51
text_pt() (text.SingleEngine method), 49
text_pt() (text.UnicodeEngine method), 59
texttextbox_pt (class in text), 52
tick (class in graph.axis.tick), 81
tickdirection() (graph.axis.axis.anchoredaxis
 method), 80
ticklength (class in graph.axis.painter), 84
tickpoint() (graph.axis.axis.anchoredaxis method), 80
tickpoint_pt() (graph.axis.axis.anchoredaxis
 method), 80
tile() (in module box), 89
tile_pt() (in module box), 89
timeaxis (class in graph.axis.timeaxis), 81
timetexter (class in graph.axis.timeaxis), 81
timetick (class in graph.axis.timeaxis), 81
tiny (text.size attribute), 58
top (text.valign attribute), 57
topzero (text.vshift attribute), 58
trafo
 module, 115

trafo (canvas.canvas attribute), 40
trafo (class in trafo), 117
trafo() (path.path method), 27
trafo_pt() (path.path method), 27
transform() (box.polygon method), 87
transformed() (path.path method), 27
translate (class in trafo), 119
translate_pt (class in trafo), 119
translated() (trafo.trafo method), 118
transparency (class in color), 109
twolines (class in connector), 92

U

UnicodeEngine (class in text), 59
unit
 module, 112
utils
 module, 127

V

valign (class in text), 57
values (class in graph.data), 70
vangle() (graph.graph.graphxyz method), 69

vbasepath() (*graph.axis.axis.anchoredaxis method*), 80 yvtickdirection() (*graphxy method*), 68
 vbasepath() (*graph.axis.positioner._positioner method*), 86
 vgeodesic() (*graph.graph.graphxy method*), 68
 vgeodesic_el() (*graph.graph.graphxy method*), 68
 vgridpath() (*graph.axis.axis.anchoredaxis method*), 80
 vgridpath() (*graph.axis.positioner._positioner method*), 86
 vpos() (*graph.graph.graphxy method*), 68
 vpos_pt() (*graph.graph.graphxy method*), 68
 vshift (*class in text*), 57
 vtickdirection() (*graph.axis.axis.anchoredaxis method*), 80
 vtickdirection() (*graph.axis.positioner._positioner method*), 86
 vtickpoint() (*graph.axis.axis.anchoredaxis method*), 80
 vtickpoint_pt() (*graph.axis.axis.anchoredaxis method*), 80
 vtickpoint_pt() (*graph.axis.positioner._positioner method*), 86
 vzindex() (*graph.graph.graphxyz method*), 69

W

warn() (*text.texmessage static method*), 55
 width (*text.texttextbox_pt attribute*), 52
 writeEPSfile() (*document.document method*), 44
 writeEPSfile() (*in module canvas*), 41
 writePDFfile() (*document.document method*), 44
 writePDFfile() (*in module canvas*), 41
 writePSfile() (*document.document method*), 44
 writePSfile() (*in module canvas*), 41
 writeSVGfile() (*document.document method*), 44
 writeSVGfile() (*in module canvas*), 41
 writetofile() (*document.document method*), 44
 writetofile() (*in module canvas*), 41

X

xbasepath() (*graphxy method*), 68
 xgridpath() (*graphxy method*), 68
 xtickdirection() (*graphxy method*), 68
 xtickpoint() (*graphxy method*), 68
 xvbasepath() (*graphxy method*), 68
 xvgridpath() (*graphxy method*), 68
 xvtickdirection() (*graphxy method*), 68
 xvtickpoint() (*graphxy method*), 68

Y

ybasepath() (*graphxy method*), 68
 ygridpath() (*graphxy method*), 68
 ytickdirection() (*graphxy method*), 68
 ytickpoint() (*graphxy method*), 68
 yvbasepath() (*graphxy method*), 68
 yvgridpath() (*graphxy method*), 68