PyX Reference Manual

Release 0.6.3

Jörg Lehmann
André Wobst

April 27, 2004

http://pyx.sourceforge.net/

Abstract

PyX is a Python package to create encapsulated PostScript figures. It provides classes and methods to access basic PostScript functionality at an abstract level. At the same time the emerging structures are very convenient to produce all kinds of drawings in a non-interactive way. In combination with the Python language itself the user can just code any complexity of the figure wanted. Additionally an TeX/LaTeX interface enables one to use the famous high quality typesetting within the figures.

A major part of PyX on top of the already described basis is the provision of high level functionality for complex tasks like 2d plots in publication-ready quality.

CONTENTS

CHAPTER

ONE

Introduction

PyX is a Python package for the creation of vector drawings. As such it allows one to readily generate encapsulated PostScript files by providing an abstraction of the PostScript graphics model. Based on this layer and in combination with the full power of the Python language itself, the user can just code any complexity of the figure wanted. PyX distinguishes itself from other similar solution by its TeX/LaTeX interface that enables one to make directly use of the famous high quality typesetting of these programs.

A major part of PyX on top of the already described basis is the provision of high level functionality for complex tasks like 2d plots in publication-ready quality.

1.1 Organisation of the PyX package

The PyX package is split in several modules, which can be categorised in the following groups

Functionality	**Modules**
basic graphics functionality	`canvas`, `path`, `deco`, `style`, `color`, and `connector`
text output via TeX/LaTeX	`text` and `box`
linear transformations and units	`trafo` and `unit`
graph plotting functionality	`graph` (including sub modules) and `graph.axis` (including sub modules)
EPS file inclusion	`epsfile`

These modules (and some other less import ones) are imported into the module namespace by using

```
from pyx import *
```

at the beginning of your Python program. However, in order to prevent namespace pollution, you may also simply use '`import pyx`'. Throughout this manual, we shall always assume that former import line form has been used.

CHAPTER

TWO

Basic graphics

2.1 Introduction

The path module allows one to construct PostScript-like *paths*, which are one of the main building blocks for the generation of drawings. A PostScript path is an arbitrary shape built up of straight lines, arc segments and cubic Bezier curves. Such a path does not have to be connected but may also consist of multiple connected segments, which will be called *sub paths* in the following.

Usually, a path is constructed by passing a list of the path primitives `moveto`, `lineto`, `curveto`, etc., to the constructor of the `path` class. The following code snippet, for instance, defines a path *p* that consists of a straight line from the point $(0, 0)$ to the point $(1, 1)$

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

Equivalently, one can also use the predefined `path` subclass `line` and write

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

While you can already do some geometrical operations with the just created path (see next section), we need another PyX object in order to be actually able to draw the path, namely an instance of the `canvas` class. By convention, we use the name *c* for this instance:

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

In order to draw the path on the canvas, we use the `stroke()` method of the `canvas` class, i.e.,

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

To complete the example, we have added a `writeEPSfile()` call, which writes the contents of the canvas into the given file.

Let us as second example define a path which consists of more than one sub path:

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

The first sub path is again a straight line from $(0, 0)$ to $(1, 1)$, with the only difference that we now have used the `rlineto` class, whose arguments count relative from the last point in the path. The second `moveto` instance opens a new sub path starting at the point $(1, 0)$ and ending at $(0, 1)$. Note that although both lines intersect at the

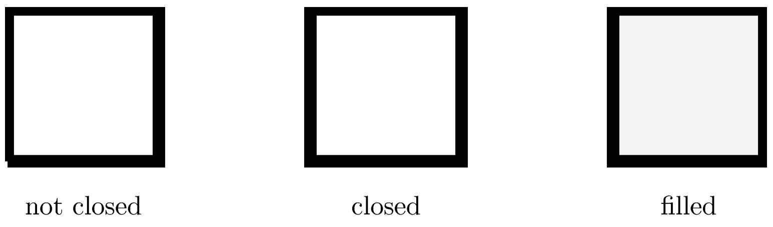


Figure 2.1: Not closed (left) and closed (midlle) rectangle. Filling a path (right) always closes it automatically.

point $(1/2, 1/2)$, they count as separate sub paths. The general rule is that each occurence of `moveto` opens a new sub path. This means that if one wants to draw a rectangle, one should not use

```
# wrong: do not use moveto when you want a single sub path
rect1 = path.path(path.moveto(0, 0), path.lineto(0, 1),
                  path.moveto(1, 0), path.lineto(1, 1),
                  path.moveto(1, 1), path.lineto(1, 1),
                  path.moveto(0, 1), path.lineto(0, 0))
```

which would construct a rectangle consisting of four disconnected sub paths. Instead the correct way of defining a rectangle is

```
# correct: a rectangle consisting of a single closed sub path
rect2 = path.path(path.moveto(0, 0), path.lineto(0, 1),
                  path.lineto(1, 1), path.lineto(1, 0),
                  path.closepath())
```

Note that for the last straight line of the rectangle (from $(0, 1)$ back to the origin at $(0, 0)$)) we have used `closepath`. This directive adds a straight line from the current point to the first point of the current sub path and furthermore *closes* the sub path, i.e., it joins the beginning and the end of the line segment. The difference can be appreciated in Fig. 2.1, where also a filled (and at the same time stroked) rectangle is shown. The corresponding code looks like

```
c.stroke(rect1, [deco.filled([color.grey(0.95)])])
```

The important point to remember here is that when filling a path, PostScript automatically closes it. More details on the available path elements can be found in Sect. 2.4.2.

XXX more on styles and attributes and reference to corresponding section

Of course, rectangles are also predefined in PyX, so above we could have as well written

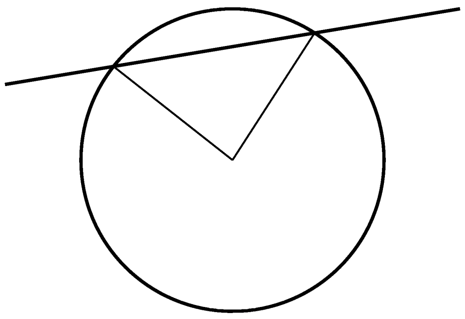
```
rect2 = path.rect(0, 0, 1, 1)
```

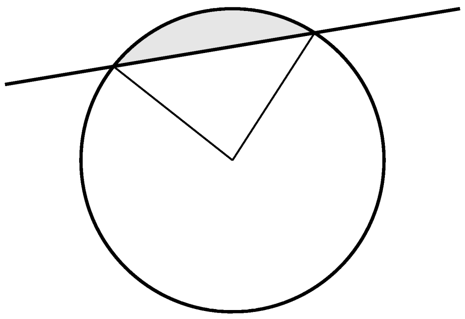
Here, the first two arguments specify the origin of the rectangle while the second two arguments define its width and height, respectively. For more details on the predefined paths, we refer the reader to Sect. 2.4.4.

2.2 Path operations

Often, one not only wants to stroke or fill a path on the canvas but before do some geometrical operations with it. For instance, one might want to intersect one path with another one and the split the paths at the intersection points and then join the segments together in a new way. PyX supports such tasks by means of a number of path methods, which we will introduce in the following.

Suppose you want to draw the radii to the intersection points of a circle with a straight line. This task can be done





using the following code which gives the result shown in Fig. 2.2

```
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:
    c.stroke(path.line(0, 0, *circle.at(isect)))

c.writeEPSfile("radii")
```

Passing another path, here *line*, to the `intersect()` method of *circle*, we obtain a tuple of parameter values of the intersection points. The first element of the tuple is a list of parameter values for the path whose `intersect()` method we have called, the second element is the corresponding list for the path passed as argument to this method. In the present example, we only need one list of parameter values, namely *isects_circle*. Iterating over the elements of this list, we draw the radii, using the `at()` path method to obtain the point corresponding to the parameter value.

Another powerful feature of PyX is its ability to split paths at a given set of parameters. For instance, in order to fill in the previous example the segment of the circle delimited by the straight line (cf. Fig. 2.2), you first have to construct a path corresponding to the outline of this segment. The following code snippet does yield this *segment*

```
arc1, arc2 = circle.split(isects_circle)
arc = arc1.arclen()<arc2.arclen() and arc1 or arc2

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

segment = line2 << arc
```

Here, we first split the circle using the `split()` method passing the list of parameters obtained above. Since the circle is closed, this yields two arc segments. We then use the `arclen()`, which returns the arc length of the path, to find the shorter of the two arcs. Before splitting the line, we have to take into account that the `split()` method only accepts a sorted list of parameters. Finally, we join the straight line and the arc segment. For this, we make use of the `<<` operator, which not only adds the paths (which could be done using '`line2 + arc`'), but also joins the last sub path of *line2* and the first one of *arc*. Thus, *segment* consists of only a single sub path and filling works as expected.

An important issue when operating on paths is the parametrisation used. Internally, PyX uses a parametrisation which uses an interval of length 1 for each path element of a path. For instance, for a simple straight line, the possible parameter values range from 0 to 1, corresponding to the first and last point, respectively, of the line. Appending another straight line, would extend this range to a maximal value of 2. You can always query this maximal value using the `range()` method of the `path` class.

However, the situation becomes more complicated if more complex objects like a circle are involved. Then, one could be tempted to assume that again the parameter value range from 0 to 1, because the predefined circle consists just of one `arc` together with a `closepath` element. However, as a simple '`path.circle(0, 0, 1).range()`' will tell, this is not the case: the actual range is much larger. The reason for this behaviour lies in the internal path handling of PyX: Before performing any non-trivial geometrical operation with a path, it will automatically be converted into an instance of the `normpath` class (see also Sect. 2.4.3). These so generated paths are already separated in their sub paths and only contain straight lines and Bézier curve segments. Thus, as is easily imaginable, they are much simpler to deal with.

A unique way of accessing a point on the path is to use the arc length of the path segment from the first point of the path to the given point. Thus, all PyX path methods that accept a parameter value also allow the user to pass an arc length. For instance,

```
from math import pi

pt1 = path.circle(0, 0, 1).at(arclen=pi)
pt2 = path.circle(0, 0, 1).at(arclen=3*pi/2)

c.stroke(path.path(path.moveto(*pt1), path.lineto(*pt2)))
```

will draw a straight line from a point at angle 180 degrees (in radians π) to another point at angle 270 degrees (in radians $3\pi/2$) on the unit circle.

More information on the available path methods can be found in Sect. 2.4.1.

2.3 Attributes: Styles and Decorations

XXX to be done

2.4 Module `path`

The `path` module defines several important classes which are documented in the present section.

2.4.1 Class `path` — PostScript-like paths

class `path`(**pathels*)

This class represents a PostScript like path consisting of the path elements *pathels*.

All possible path elements are described in Sect. 2.4.2. Note that there are restrictions on the first path element and likewise on each path element after a `closepath` directive. In both cases, no current point is defined and the path element has to be an instance of one of the following classes: `moveto`, `arc`, and `arcn`.

Instances of the class `path` provide the following methods (in alphabetic order):

`append`(*pathel*)

Appends a *pathel* to the end of the path.

`arclen`( )

Returns the total arc length of the path.†

`arclentoparam`(*lengths*)

Returns the parameter values corresponding to the arc lengths *lengths*.†

`at`(*param=None, arclen=None*)

Returns the coordinates (as 2-tuple) of the path point corresponding to the parameter value *param* or, alternatively, the arc length *arclen*. The parameter value *param* (*arclen*) has to be smaller or equal to `self.range()` (`self.arclen()`), otherwise an exception is raised. At discontinuities in the path, the limit from below is returned.†

`bbox`( )

Returns the bounding box of the path. Note that this returned bounding box may be too large, if the path contains any `curveto` elements, since for these the control box, i.e., the bounding box enclosing the control points of the Bézier curve is returned.

`begin`( )

Returns the coordinates (as 2-tuple) of the first point of the path.†

`curvradius`(*param=None, arclen=None*)

Returns the curvature radius (or None if infinite) at parameter param or, alternatively, arc length *arclen*. This is the inverse of the curvature at this parameter Please note that this radius can be negative or positive, depending on the sign of the curvature.†

`end`( )

Returns the coordinates (as 2-tuple) of the end point of the path.†

`intersect`(*opath*)

Returns a tuple consisting of two lists of parameter values corresponding to the intersection points of the path with the other path *opath*, respectively.†

`joined`(*opath*)

Appends *opath* to the end of the path, thereby merging the last sub path (which must not be closed) of the path with the first sub path of *opath* and returns the resulting new path.†

`range`( )

Returns the maximal parameter value *param* that is allowed in the path methods.

`reversed`( )

Returns the reversed path.†

`split`(*params*)

Splits the path at the parameters *params*, which have to be sorted in ascending order, and returns a corresponding list of `normpath` instances.†

`tangent`(*param=None, arclen=None, length=None*)

Return a `line` instance corresponding to the tangent vector to the path at the parameter value *param* or, alternatively, the arc length *arclen*. The parameter value *param* (*arclen*) has to be smaller or equal to `self.range()` (`self.arclen()`), otherwise an exception is raised. At discontinuities in the path, the limit from below is returned. If *length* is not `None`, the tangent vector will be scaled correspondingly.†

`trafo`(*param=None, arclen=None*)

Returns a trafo which maps a point $(0, 1)$ to the tangent vector to the path at the parameter value *param* or, alternatively, the arc length *arclen*. The parameter value *param* (*arclen*) has to be smaller or equal to `self.range()` (`self.arclen()`), otherwise an exception is raised. At discontinuities in the path, the limit from below is returned.†

`transformed`(*trafo*)

Returns the path transformed according to the linear transformation *trafo*. Here, `trafo` must be an instance of the `trafo.trafo` class.†

Some notes on the above:

- The † denotes methods which require a prior conversion of the path into a `normpath` instance. This is done automatically, but if you need to call such methods often or if you need to change the precision used for this conversion, it is a good idea to manually perform the conversion.
- Instead of using the `joined()` method, you can also join two paths together with help of the `<<` operator, for instance '`p = p1 « p2`'.
- In the methods accepting both a parameter value *param* and an arc length *arclen*, exactly one of these arguments has to provided.

2.4.2 Path elements

The class `pathel` is the superclass of all PostScript path construction primitives. It is never used directly, but only by instantiating its subclasses, which correspond one by one to the PostScript primitives.

Except for the path elements ending in `_pt`, all coordinates passed to the path elements can be given as number (in which case they are interpreted as user units with the currently set default type) or in PyX lengths.

The following operation move the current point and open a new sub path:

class `moveto`(*x, y*)

Path element which sets the current point to the absolute coordinates (*x*, *y*). This operation opens a new subpath.

class `rmoveto`(*dx, dy*)

Path element which moves the current point by (*dx*, *dy*). This operation opens a new subpath.

Drawing a straight line can be accomplished using:

class `lineto`(*x, y*)

Path element which appends a straight line from the current point to the point with absolute coordinates (*x*, *y*), which becomes the new current point.

class `rlineto`(*dx, dy*)

Path element which appends a straight line from the current point to the a point with relative coordinates (*dx*, *dy*), which becomes the new current point.

For the construction of arc segments, the following three operations are available:

class `arc`(*x, y, r, angle1, angle2*)

Path element which appends an arc segment in counterclockwise direction with absolute coordinates (*x*, *y*) of the center and radius *r* from *angle1* to *angle2* (in degrees). If before the operation, the current point is defined, a straight line is from the current point to the beginning of the arc segment is prepended. Otherwise, a subpath, which thus is the first one in the path, is opened. After the operation, the current point is at the end of the arc segment.

class `arcn`(*x, y, r, angle1, angle2*)
Path element which appends an arc segment in clockwise direction with absolute coordinates (*x*, *y*) of the center and radius *r* from *angle1* to *angle2* (in degrees). If before the operation, the current point is defined, a straight line is from the current point to the beginning of the arc segment is prepended. Otherwise, a subpath, which thus is the first one in the path, is opened. After the operation, the current point is at the end of the arc segment.

class `arct`(*x1, y1, x2, y2, r*)
Path element which appends an arc segment of radius *r* connecting between (*x1*, *y1*) and (*x2*, *y2*).

Bézier curves can be constructed using:

class `curveto`(*x1, y1, x2, y2, x3, y3*)
Path element which appends a Bézier curve with the current point as first control point and the other control points (*x1*, *y1*), (*x2*, *y2*), and (*x3*, *y3*).

class `rcurveto`(*dx1, dy1, dx2, dy2, dx3, dy3*)
Path element which appends a Bézier curve with the current point as first control point and the other control points defined relative to the current point by the coordinates (*dx1*, *dy1*), (*dx2*, *dy2*), and (*dx3*, *dy3*).

Note that when calculating the bounding box (see Sect. 9) of Bézier curves, PyX uses for performance reasons the so-called control box, i.e., the smallest rectangle enclosing the four control points of the Bézier curve. In general, this is not the smallest rectangle enclosing the Bézier curve.

Finally, an open sub path can be closed using:

class `closepath`()
Path element which closes the current subpath.

For performance reasons, two non-PostScript path elements are defined, which perform multiple identical operations:

class `multilineto_pt`(*points*)
Path element which appends straight line segments starting from the current point and going through the list of points given in the *points* argument. All coordinates have to be given in PostScript points.

class `multicurveto_pt`(*points*)
Path element which appends Bézier curve segments starting from the current point and going through the list of each three control points given in the *points* argument.

2.4.3 Class `normpath`

The `normpath` class represents a specialized form of a `path` containing only the elements `moveto`, `lineto`, `curveto` and `closepath`. Such normalized paths are used for all of the more sophisticated path operations which are denoted by a † in the description of the `path` class above.

Any path can easily be converted to its normalized form by passing it as parameter to the `normpath` constructor,

```
np = normpath(p)
```

Additionally, you can specify the accuracy (in points) which is used in all `normpath` calculations by means of the keyword argument *epsilon*, which defaults to 10^{-5}. Note that the sum of a `normpath` and a `path` always yields a `normpath`.

class `normpath`(*arg=[], epsilon=1e-5*)
Construct a normpath from *arg*. All numerical calculations will be performed using an accuracy of the order of *epsilon* points. The first argument *arg* can be a `path` or a another `normpath` instance. Alternatively, a list of `normsubpath` instances can be supplied as argument.

In addition to the `path` methods, a `normpath` instance also offers the following methods, which operate on the instance itself:

`join`(*other*)

Join *other*, which has to be a `path` instance, to the `normpath` instance.

`reverse``()`
Reverses the `normpath` instance.

`transform``(`*trafo*`)`
Transforms the `normpath` instance according to the linear transformation *trafo*.

2.4.4 Predefined paths

For your convenience, some oft-used paths are already pre-defined. All of them are sub classes of the `path` class.

class `line``(`*x1, y1, x2, y2, x3, y3*`)`
A straight line from the point (*x1*, *y1*) to the point (*x2*, *y2*).

class `curve``(`*x1, y1, x2, y2, x3, y3, x4, y4*`)`
A Bézier curve with control points (*x0*, *y0*), . . ., (*x3*, *y3*).

class `rect``(`*x, y, w, h*`)`
A closed rectangle with lower left point (*x*, *y*), width *w*, and height *h*.

class `circle``(`*x, y, r*`)`
A closed circle with center (*x*, *y*) and radius *r*.

2.5 Module canvas

One of the central modules for the PostScript access in PyX is named `canvas`. Besides providing the class `canvas`, which presents a collection of visual elements like paths, other canvases, TeX or LaTeX elements, it contains the class `canvas.clip` which allows clipping of the output.

A canvas may also be embedded in another one using its `insert` method. This may be useful when you want to apply a transformation on a whole set of operations..

2.5.1 Class canvas

This is the basic class of the canvas module, which serves to collect various graphical and text elements you want to write eventually to an (E)PS file.

class **`canvas`**(*attrs=[], texrunner=None*)

Construct a new canvas, applying the given *attrs*, which can be instances of `trafo.trafo`, `canvas.clip`, `style.strokestyle` or `stlye.fillstyle`. The *texrunner* argument can be used to specify the texrunner instance used for the `text()` method of the canvas. If not specified, it defaults to *text.defaulttexrunner*.

Paths can be drawn on the canvas using one of the following methods:

`draw`(*path, attrs*)

Draws *path* on the canvas applying the given *attrs*.

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

Fills the given *path* on the canvas applying the given *attrs*.

`stroke`(*path, attrs=[]*)

Strokes the given *path* on the canvas applying the given *attrs*.

Arbitrary allowed elements like other `canvas` instances can be inserted in the canvas using

`insert`(*PSOp, attrs=[]*)

Inserts an instance of `base.PSOp` into the canvas. If *attrs* are present, PSOp is inserted into a new `canvas`instance with *attrs* as arguments passed to its constructor is created. Then this `canvas` instance is inserted itself into the canvas. Returns *PSOp*.

Text output on the canvas is possible using

`text`(*x, y, text, attrs=[]*)

Inserts *text* at position (*x*, *y*) into the canvas applying *attrs*. This is a shortcut for `insert(texrunner.text(x, y, text, attrs))`).

The `canvas` class provides access to the total geometrical size of its element:

`bbox`()

Returns the bounding box enclosing all elements of the canvas.

A canvas also allows one to set global options:

`set`(*styles*)

Sets the given *styles* (instances of `style.fillstyle` or `style.strokestyle` or subclasses thereof). for the rest of the canvas.

`settexrunner`(*texrunner*)

Sets a new *texrunner* for the canvas.

The contents of the canvas can be written using:

`writeEPSfile`(*filename, paperformat=None, rotated=0, fittosize=0, margin="1 t cm", bbox=None, bboxenlarge="1 t pt"*)

Writes the canvas to *filename* (the extension `.eps` is appended automatically). Optionally, a *paperformat* can be specified, in which case the output will be centered with respect to the corresponding size using the given *margin*. See *canvas._paperformats* for a list of known paper formats . Use *rotated*, if you want to center on a 90° rotated version of the respective paper format. If *fittosize* is set, the output is additionally scaled to the maximal possible size. Normally, the bounding box of the canvas is calculated automatically from the bounding box of its elements. Alternatively, you may specify the *bbox* manually. In any case, the bounding box becomes enlarged on all side by *bboxenlarge*. This may be used to compensate for the inability of PyX to take the linewidths into account for the calculation of the bounding box.

2.5.2 Patterns

The `pattern` class allows the definition of PostScript Tiling patterns (cf. Sect. 4.9 of the PostScript Language Reference Manual) which may then be used to fill paths. The classes `pattern` and `canvas` differ only in their constructor and in the absence of a `writeEPSfile()` method in the former. The `pattern` constructor accepts the following keyword arguments:

keyword	description
`painttype`	`1` (default) for coloured patterns or `2` for uncoloured patterns
`tilingtype`	`1` (default) for constant spacing tilings (patterns are spaced constantly by a multiple of a device pixel), `2` for undistored pattern cell, whereby the spacing may vary by as much as one device pixel, or `3` for constant spacing and faster tiling which behaves as tiling type `1` but with additional distortion allowed to permit a more efficient implementation.
`xstep`	desired horizontal spacing between pattern cells, use `None` (default) for automatic calculation from pattern bounding box.
`ystep`	desired vertical spacing between pattern cells, use `None` (default) for automatic calculation from pattern bounding box.
`bbox`	bounding box of pattern. Use `None` for an automatical determination of the bounding box (including an enlargement by 5 pts on each side.)
`trafo`	additional transformation applied to pattern or `None` (default). This may be used to rotate the pattern or to shift its phase (by a translation).

After you have created a pattern instance, you define the pattern shape by drawing in it like in an ordinary canvas. To use the pattern, you simply pass the pattern instance to a `stroke()`, `fill()`, `draw()` or `set()` method of the canvas, just like you would do with a colour, etc.

CHAPTER

THREE

Module `text`: TEX/LATEX interface

3.1 Basic functionality

The `text` module seamlessly integrates the famous typesetting technique of TEX/LATEX into PYX. The basic procedure is:

- start a TEX/LATEX instance as soon as an TEX/LATEX preamble setting or a text creation is requested
- create boxes containing the requested text and shipout those boxes to the dvi file
- immediately analyse the TEX/LATEX output for errors; the box extents are also contained in the TEX/LATEX output and thus become available immediately
- when your TeX installation supports the `ipc` mode and PYX is configured to use it, the dvi output is also analysed immediately; alternatively PYX quits the TEX/LATEX instance to read the dvi output once PostScript needs to be written or markers are accessed
- Type1 fonts are used for the PostScript generation

Note that for using Type1 fonts an appropriate font mapping file has to be provided. When your TEX installation is configured to use Type1 fonts by default, the `psfonts.map` will contain entries for the standard TEX fonts already. Alternatively, you may either look for updmap used by many TEX installations to create an appropriate font mapping file or you may specify some alternative font mapping files like `psfonts.cmz` in the `pyxrc` or the `fontmaps` keyword argument of the `texrunner` constructor (or the `set` method).

3.2 The texrunner

Instances of the class `texrunner` represent a TEX/LATEX instance. The keyword arguments of the constructor are listed in the following table:

keyword	description
`mode`	`"tex"` (default) or `"latex"`
`lfs`	Specifies a latex font size file to be used with TEX (not in LATEX). Those files (with the suffix `.lfs`) can be created by `createlfs.tex`. Possible values are listed when a requested name could not be found.
`docclass`	LATEX document class; default is `"article"`
`docopt`	specifies options for the document class; default is `None`
`usefiles`	access to TEX/LATEX jobname files; default: `None`; example: `["spam.aux", "eggs.log"]`
`fontmaps`	whitespace separated names of font mapping files; default `"psfonts.map"`
`waitfortex`	wait this number of seconds for a TEX/LATEX response; default `60`
`showwaitfortex`	show a message about waiting for TEX/LATEX response on `strerr`; default `5`
`texipc`	use the `-ipc` option of TEX/LATEX for immediate dvi-output access (boolean); check the output of `tex -help` if this option is available in your TEX/LATEX installation; default `0`
`texdebug`	filename to store TEX/LATEX commands; default `None`
`dvidebug`	dvi debug messages like `dvitype` (boolean); default `0`
`errordebug`	verbose level of TEX/LATEX error messages; valid values are `0`, `1` (default), `2`
`pyxgraphics`	enables the usage of the graphics package without further configuration (boolean); default 1
`texmessagesstart`	parsers for the TEX/LATEX start message; default: `[texmessage.start]`
`texmessagesdocclass`	parsers for LATEXs `\documentclass` statement; default: `[texmessage.load]`
`texmessagesbegindoc`	parsers for LATEXs `\begin{document}` statement; default: `[texmessage.load, texmessage.noaux]`
`texmessagesend`	parsers for TEXs `\end`/ LATEXs `\end{document}` statement; default: `[texmessage.texend]`
`texmessagesdefaultpreamble`	default parsers for preamble statements; default: `[texmessage.load]`
`texmessagesdefaultrun`	default parsers for text statements; default: `[texmessage.loadfd, texmessage.graphicsload]`

The default values of the parameters `fontmaps`, `waitfortex`, `showwaitfortex`, and `texipc` can be modified in the `text` section of a `pyxrc`.

The `texrunner` instance provides several methods to be called by the user. First there is a method called `set`. It takes the same kewword arguments as the constructor and its purpose is to provide an access to the `texrunner` settings for a given instance. This is important for the `defaulttextunner`. The `set` method fails, when a modification can't be applied anymore (e.g. TEX/LATEX was already started).

The `preamble` method can be called before the `text` method only (see below). It takes a TEX/LATEX expression and optionally a list of TEX/LATEX message parsers. The preamble expressions should be used to perform global settings, but should not create any TEX/LATEX dvi output. In LATEX, the preamble expressions are inserted before the `\begin{document}` statement. Note, that you can use `\AtBeginDocument{...}` to postpone the direct evaluation.

Finally there is a `text` method. The first two parameters are the `x` and `y` position of the output to be generated. The third parameter is a TEX/LATEX expression. There are two further keyword arguments. The first, `textattrs`, is a list of TEX/LATEX settings as described below, PYX transformations, and PYX fill styles (like colors). The second keyword argument `texmessages` takes a list of TEX/LATEX message parsers as described below as well. The `text` method returns a box (see chapter 6), which can be inserted into a canvas instance by its `insert` method to get the text.

The box returned by the `text` method has an additional method `marker`. You can place markers in the TEX/LATEX expression by the command `\PyXMarker{<string>}`. When calling the `marker` method with the same

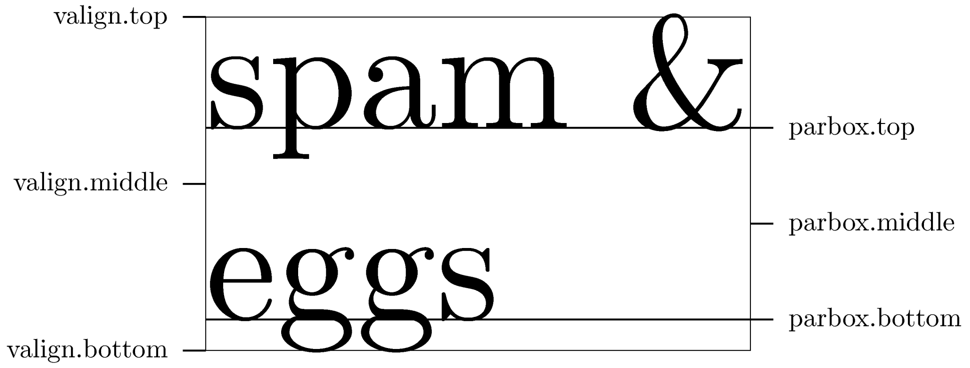


Figure 3.1: valign example

`<string>` you can get back the position of the marker later on. Only digits, letters and the `@` symbol are allowed within the string. Strings containing the `@` symbol are not considered for end users like it is done for commands including the `@` symbol in LaTeX.

Note that for the generation of the PostScript code the TeX/LaTeX instance must be terminated except when `texipc` is turned on. However, a TeX/LaTeX instance is started again when the `text` method is called again. A call of the `preamble` method will still fail, but you can explicitly call the `reset` method to allow for new `preamble` settings as well. The `reset` method takes a boolean parameter `reinit` which can be set to run the old preamble settings.

3.3 TeX/LaTeX attributes

Horizontal alignment: `halign.left` (default), `halign.center`, `halign.right`, `halign(x)` (`x` is a value between `0` and `1` standing for left and right, respectively)

Vertical alignment: `valign.top`, `valign.middle`, `valign.bottom`, `valign.baseline` (default); see the left hand side of figure 3.1

Vertical box: Usually, TeX/LaTeX expressions are handled in horizontal mode (so-called LR-mode in TeX/LaTeX; everything goes into a single line). You may use `parbox(x)`, where `x` is the width of the text, to switch to a multiline mode (so-called vertical mode in TeX/LaTeX). The additional keyword parameter `baseline` allows the user to alter the position of the baseline. It can be set to `parbox.top` (default), `parbox.middle`, `parbox.bottom` (see the right hand side of figure 3.1). The baseline position is relevant when the vertical alignment is set to baseline only.

Vertical shift: `vshift(lowerratio, heightstr="0")` (lowers the output by `lowerratio` of the height of `heightstr`), `vshift.bottomzero=vshift(0)` (doesn't have an effect), `vshift.middlezero=vshift(0.5)` (shifts down by half of the height of `0`), `vshift.topzero=vshift(1)` (shifts down by the height of `0`), `vshift.mathaxis` (shifts down by the height of the mathematical axis)

Mathmode: `mathmode` switches to mathmode of TeX/LaTeX in `\displaystyle` (`nomathmode` removes this attribute)

Font size: `size.tiny=size(-4)`, `size.scriptsize=size(-3)`, `size.footnotesize=size(-2)`, `size.small=size(-1)`, `size.normalsize=size(0)`, (default), `size.large=size(1)`, `size.Large=size(2)`, `size.LARGE=size(3)`, `size.huge=size(4)`, `size.Huge=size(5)`

3.4 Using the graphics-bundle with LaTeX

The packages in LaTeX-graphics bundle (color.sty, graphics.sty, graphicx.sty, ...) make extensive use of `\special` commands. Here are some notes on this topic. Please install the appropriate driver file `pyx.def`, which defines all the specials, in your LaTeX-tree and add the content of both files `color.cfg` and `graphics.cfg` to your personal configuration files.[1] After you have installed the `.cfg` files please use the `text` module always with the `pyxgraphics` keyword set to 0, this switches off a hack that might be convenient for less experienced LaTeX-users.

You can then import the packages of the graphics-bundle and related packages (e.g. rotating, …) with the option `pyx`, e.g. `\usepackage[pyx]{color,graphicx}`. Please note that the option `pyx` is only available with `pyxgraphics=0` and a properly installed driver file. Otherwise do not use this option, omit it completely or say `[dvips]`.

When defining colours in LaTeX as one of the colour models `gray`, `cmyk`, `rgb`, `RGB`, `hsb` then PyX will use the corresponding values (one to four real numbers) for output. When you use one of the `named` colors in LaTeX then PyX will use the corresponding predefined colour (see module `color` and the colour table at the end of the manual).

When importing eps-graphics in LaTeX then PyX will rotate, scale and clip your file like you expect it. Note that PyX cannot import other graphics files than `eps` at the moment.

For reference purpose, the following specials can be handled by the `text` module at the moment:

`PyX:color_begin (model) (spec)`
: starts a colour. (model) is one of {`gray`, `cmyk`, `rgb`, `hsb`, `texnamed`}. (spec) depends on the model: a name or some numbers.

`PyX:color_end` ends a colour.

`PyX:epsinclude file= llx= lly= urx= ury= width= height= clip=0/1`
: includes an eps-file. The values of llx to ury are in the files' coordinate system and specify the part of the graphics that should become the specified width and height in the outcome. The graphics may be clipped. The last three parameters are optional.

`PyX:scale_begin (x) (y)`
: begins scaling from the current point.

`PyX:scale_end` ends scaling.

`PyX:rotate_begin (angle)` begins rotation around the current point.

`PyX:rotate_end` ends rotation.

3.5 TeX/LaTeX message parsers

Message parsers are used to scan the output of TeX/LaTeX. The output is analysed by a sequence of message parsers. Each of them analyses the output and remove those parts of the output, it feels responsible for. If there is nothing left in the end, the message got validated, otherwise an exception is raised reporting the problem.

parser name	purpose
`texmessage.load`	loading of files (accept `(file ...)`)
`texmessage.loadfd`	loading of files (accept `(file.fd)`)
`texmessage.graphicsload`	loading of graphic files (accept `<file.eps>`)
`texmessage.ignore`	accept everything as a valid output

More specialised message parsers should become available as required. Please feal free to contribute (e.g. with ideas/problems; code is desired as well, of course). There are further message parsers for PyXs internal use, but we skip them here as they are not interesting from the users point of view.

3.6 The defaulttexrunner instance

The `defaulttexrunner` is an instance of the class `texrunner`, which is automatically created by the `text` module. Additionally, the methods `text`, `preamble`, and `set` are available as module functions accessing the `defaulttexrunner`. This single `texrunner` instance is sufficient in most cases.

[1]If you do not know what I am talking about right now – just ignore this paragraph, but make sure not to set the `pyxgraphics` keyword to 0.

CHAPTER

FOUR

Graphs

4.1 Introduction

PyX can be used for data and function plotting. At present only x-y-graphs are supported. However, the component architecture of the graph system described in section 4.2 allows for additional graph geometries while reusing most of the existing components.

Creating a graph splits into two basic steps. First you have to create a graph instance. The most simples form would look like:

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

The graph instance `g` created in this example can than be used to actually plot something into the graph. Suppose you have some data in a file 'graph.dat' you want to plot. The content of the file could look like:

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

To plot these data into the graph `g` you must perform:

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

The method `plot()` takes the data to be plotted and optionally a graph style to be used to plot the data. When no style is provided, a default style defined by the data instance is used. For data read from a file by an instance of `graph.data.file`, the default are symbols. When instantiating `graph.data.file`, you not only specify the file name, but also a mapping from columns to axis names and other information the style might use (*e.g.* data for error bars for the symbol style).

While the graph is already created by that, we still need to perform a write of the result into a file. Since the graph instance is a canvas, we can just call its `writeEPSfile()` method.

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

will create the file 'graph.eps' as shown in figure 4.1.

Instead of plotting data from a file, we could also use other data sources like functions. The procedure would be as before, but we would place different data into `plot()`:

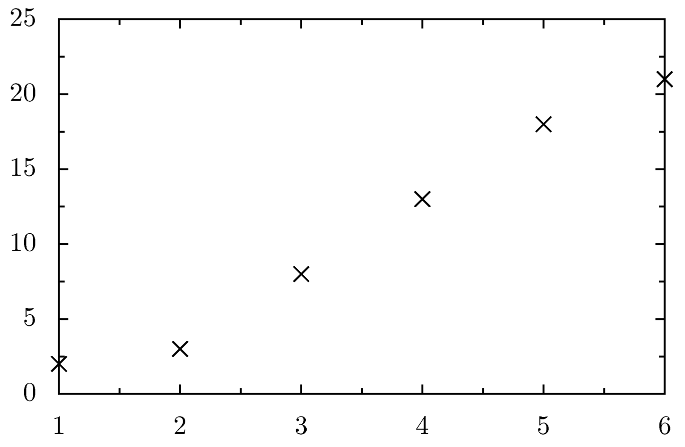


Figure 4.1: A minimalistic plot for the data from file 'graph.dat'.

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

You can plot different data into a single graph by calling the `plot()` several times. Thus the command above might just be inserted before `writeEPSfile()` of the original example. Note that a call to `plot()` will fail once you forced the graph to "finish" itself. This happens automatically, when you write the output. Thus it is not an option to call `plot()` after `writeEPSfile()`. The topic of the finalization of a graph is addressed in more detail in section 4.3. As you can see in figure 4.2, a function is plotted as a line by default.

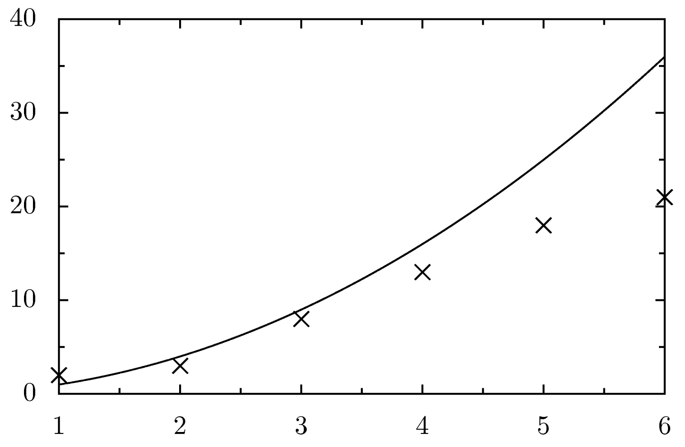


Figure 4.2: Plotting data from a file together with a function.

4.2 Component architecture

Creating a graph involves a variety of tasks, which thus can be separated into components without significant additional costs. This structure manifests itself also in the PyX source, where there are different modules for the different tasks. They interact by some welldefined interfaces. They certainly has to be completed and stabilized in its details, but the basic structure came up in the continuous development quite clearly. The basic parts of a graph are:

graph
: Defines the geometry of the graph by means of graph coordinates with range [0:1]. Keeps lists of plotted data, axes *etc.*

data

Produces or prepare data to be plotted in graphs.

style
: Performs the plotting of the data into the graph. It get data, convert them via the axes into graph coordinates and uses the graph to finally plot the data with respect to the graph geometry methods.

key
: Responsible for the graph keys.

axis
: Creates axes for the graph, which take care of the mapping from data values to graph coordinates. Because axes are also responsible for creating ticks and labels, showing up in the graph themselfs and other things, this task is splitted into several independend subtasks. Axes are discussed separately in chapter 5.

4.3 X-Y-Graphs

The class `graphxy` is part of the module `graph.graph`. However, there is a shortcut to access this class via `graph.graphxy`.

class `graphxy`(*xpos=0, ypos=0, width=None, height=None, ratio=goldenmean, key=None, backgroundattrs=None, axesdist="0.8 cm", **axes*)

This class provides a x-y-graph. A graph instance is also a full functional canvas.

The position of the graph on its own canvas is specified by *xpos* and *ypos*. The size of the graph is specified by *width*, *height*, and *ratio*. These parameters define the size of the graph area not taking into account the additional space needed for the axes. Note that you have to specify at least *width* or *height*. *ratio* will be used as the ratio between *width* and *height* when only one of these are provided.

key can be set to a `graph.key.key` instance to create an automatic graph key. `None` omits the graph key.

backgroundattrs is a list of attributes for drawing the background of the graph. Allowed are decorators, strokestyles, and fillstyles. `None` disables background drawing.

axisdist is the distance between axes drawn at the same side of a graph.

***axes* recieves axes instances. Allowed keywords (axes names) are `x`, `x2`, `x3`, *etc.* and `y`, `y2`, `y3`, *etc.* When not providing a `x` or `y` axis, linear axes instances will be used automatically. When not providing a `x2` or `y2` axis, linked axes to the `x` and `y` axes are created automatically. You may set those axes to `None` to disable the automatic creation of axes. The even numbered axes are plotted at the top (`y` axes) and right (`x` axes) while the others are plotted at the bottom (`x` axes) and left (`y` axes) in ascending order each. Axes instances should be used once only.

Some instance attributes might be usefull for outside read-access. Those are:

`axes`
: A dictionary mapping axes names to the `axis` instances.

`axespos`
: A dictionary mapping axes names to the `axispos` instances.

To actually plot something into the graph, the following instance method `plot()` is provided:

`plot`(*data, style=None*)

Adds *data* to the list of data to be plotted. Sets *style* the be used for plotting the data. When *style* is `None`, the default style for the data as provided by *data* is used.

data should be an instance of any of the data described in section 4.4. This instance should used once only.

When a style is used several times within the same graph instance, it is kindly asked by the graph to iterate its appearence. Its up to the style how this is performed.

Instead of calling the plot method several times with different *data* but the same style, you can use a list (or something iterateable) for *data*.

While a graph instance only collects data initially, at a certain point it must create the whole plot. Once this is done, further calls of `plot()` will fail. Usually you do not need to take care about the finalization of the graph, because it happens automatically once you write the plot into a file. However, sometime position methods (described below) are nice to be accessable. For that, at least the layout of the graph must be done. By calling the `do`-methods yourself you can also alter the order in which the graph is plotted. Multiple calls the any of the `do`-methods have no effect (only the first call counts). The orginal order in which the `do`-methods are called is:

`dolayout``()`
: Fixes the layout of the graph. As part of this work, the ranges of the axes are fitted to the data when the axes ranges are allowed to addjust themselfs to the data ranges. The other `do`-methods ensure, that this method is always called first.

`dobackground``()`
: Draws the background.

`doaxes``()`
: Inserts the axes.

`dodata``()`
: Plots the data.

`dokey``()`
: Inserts the graph key.

`finish``()`
: Finishes the graph by calling all pending `do`-methods. This is done automatically, when the output is created.

The graph provides some methods to access its geometry:

`pos``(`*x, y, xaxis=None, yaxis=None*`)`
: Returns the given point at *x* and *y* as a tuple `(xpos, ypos)` at the graph canvas. *x* and *y* are axis data values for the two axes *xaxis* and *yaxis*. When *xaxis* or *yaxis* are `None`, the axes with names `x` and `y` are used. This method fails if called before `dolayout()`.

`vpos``(`*vx, vy*`)`
: Returns the given point at *vx* and *vy* as a tuple `(xpos, ypos)` at the graph canvas. *vx* and *vy* are graph coordinates with range [0:1].

`vgeodesic``(`*vx1, vy1, vx2, vy2*`)`
: Returns the geodesic between points *vx1*, *vy1* and *vx1*, *vy1* as a path. All parameters are in graph coordinates with range [0:1]. For `graphxy` this is a straight line.

`vgeodesic_el``(`*vx1, vy1, vx2, vy2*`)`
: Like `vgeodesic()` but this method returns the path element to connect the two points.

Further geometry information is available by the `axespos` instance variable. Shortcuts to the `axispos` methods for the `x` and `y` axis become available after `dolayout()` as `graphxy` methods `Xbasepath`, `Xvbasepath`, `Xgridpath`, `Xvgridpath`, `Xtickpoint`, `Xvtickpoint`, `Xtickdirection`, and `Xvtickdirection` where the prefix `X` stands for `x` and `y`.

4.4 Data

The following classes provide data for the `plot()` method of a graph. The classes are implemented in `graph.data`.

class `file``(`*filename, commentpattern=defaultcommentpattern, columnpattern=defaultcolumnpattern, stringpattern=defaultstringpattern, skiphead=0, skiptail=0, every=1, title=notitle, parser=dataparser(), context={}, **columns*`)`
: This class reads data from a file and makes them available to the graph system. *filename* is the name of the file to be read. The data should be organized in columns.

  The arguments *commentpattern*, *columnpattern*, and *stringpattern* are responsible for identifying the data in each line of the file. Lines matching *commentpattern* are ignored except for the column name search of the last non-emtpy comment line before the data. By default a line starting with one of the characters '`#`', '`%`', or '`!`' are treated as comments. A line is analysed by repeatingly matching *stringpattern* and, whenever the stringpattern does not match by *columnpattern*. When the *stringpattern* matches, the result is taken as the value for the next column without further transformations. When *columnpattern* matches, it is tried to convert the result to a float. When this fails the result is taken as a string as well. By default, you can write strings with spaces surrounded by '`"`' immediately surrounded by spaces or begin/end of line in the data file. Otherwise '`"`' is not taken to be special.

skiphead and *skipfoot* are numbers of data lines to be ignored at the beginning and end of the file while *every* selects only every *every* line from the data.

title is the title of the data to be used in the graph key. A default title is constructed out of *filename* and ***columns*. You may set *title* to `None` to disable the title.

parser is the parser for mathematical expression provided in ***columns*. When in doubt, this is probably uninteresting for you. *context* allows for accessing external variables and functions when evaluating mathematical expressions for columns. As an example you may use `context=locals()` or something similar.

Finally, *columns* defines the data columns. To make it a bit more complicated, there are file column names and new created data column names, namely the keywords of ***columns*. File column names occure when the data file contains a comment line immediately in front of the data. This line will be parsed skipping the comment character (even if it occures multiple times) as if it would be regular data, but it will not be converted to floats even if it would be possible to convert them. The values of ***columns* can refer to column numbers in the file (starting with `1`). The column `0` is also available and contains the line number starting from `1` not counting comment lines. Furthermore values of ***columns* can be strings: file column names or mathematical expressions. To refer to columns within mathematical expressions you can also use file column names when they are valid variable names or by the syntax `$<number>` or even `$(<expression>)`, where `<number>` is a non-negative integer and `<expression>` a valid mathematical expression itself. For the later negative numbers count the columns from the end. Example:

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

with 'test.dat' looking like:

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

The columns with name '`a`', '`b`', '`c`' will become '`[1.234, 5.678]`', '`[1.0, 3.0]`', and '`[4.0, 10.0]`', respectively.

When creating the several data instances accessing the same file, the file is read only once. There is an inherent caching of the file contents.

For the sake of completeness the default patterns:

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

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

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

class `function`(*expression, title=notitle, min=None, max=None, points=100, parser=mathtree.parser(), context={}*)

This class creates graph data from a function. *expression* is the mathematical expression of the function. It must also contain the result variable name by assignment. Thus a typical example looks like '`y=sin(x)`'.

title is the title of the data to be used in the graph key. By default *expression* is used. You may set *title* to `None` to disable the title.

min and *max* give the range of the variable. If not set the range spans the hole axis range. The axis range might be set explicitly or implicitly by ranges of other data. *points* is the number of points for which the function is calculated. The points are lineary choosen in terms of graph coordinates.

parser is the parser for the mathematical expression. When in doubt, this is probably uninteresting for you. *context* allows for accessing external variables and functions. As an example you may use `context=locals()` or something similar.

Note when accessing external variables: When doing so, at first it renders unclear, which of the variables should be used as the dependent variable. The solution is, that there should be exactly one variable, which is a valid and used axis name. Example:

```
[graph.data.function("y=x**i", context=locals()) for i in range(1, 5)]
```

The result of this expression could just be passed to a graphs `plot()` method, since not only data instances but also lists of data instances are allowed.

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

This class creates graph data from a parametric function. *varname* is the parameter of the function. *min* and *max* give the range for that variable. *points* is the number of points for which the function is calculated. The points are choosen lineary in terms of the parameter.

expression is the mathematical expression for the parametric function. It contains an assignment of a tuple of functions to a tuple of variables.

title is the title of the data to be used in the graph key. By default *expression* is used. You may set *title* to `None` to disable the title.

parser is the parser for mathematical expression. When in doubt, this is probably uninteresting for you. *context* allows for accessing external variables and functions. As an example you may use `context=locals()` or something similar.

class `list`(*points, title="user provided list", maxcolumns=None, addlinenumbers=1, **columns*)

This class creates graph data from external provided data. *points* is a list of lines, where each line is a list of data values for the columns.

title is the title of the data to be used in the graph key.

maxcolumn is the number of columns in the points list. If set to `None`, the number of columns will be calculated by cycling through the points. Each element of *points*, *i.e.* each line, will be checked and adjusted to the number of columns.

addlinenumbers is a boolean indicating whether line numbers should be added or not. Note that the line numbers are storred in column `0`. A transformation (see `data` below) will always keep the first column. When not adding line numbers, you should be aware, that the numbering in ***columns* becomes different form the usual case, where the first column containing data (not the line number) has column number `1`.

The keywords of ***columns* become the data column names. The values are the column numbers starting from one, when *addlinenumbers* is turned on (the zeroth column is the line number then), while the column numbers starts from zero, when *addlinenumbers* is switched off.

class `data`(*data, title=notitle, parser=dataparser(), context=, **columns*)

This class provides graph data out of other graph data. *data* is the source of the data. All other paramters work like the equally called parameters in `graph.data.file`. Indeed, the later is build on top of this class by reading the file and caching its contents in a `graph.data.list` instance. The columns are then selected by creating new data out of the existing data. Note that the data itself is not copied as long as no new columns need to be calculated.

class `conffile`(*filename, title=notitle, parser=dataparser(), context=, **columns*)

This class reads data from a config file with the file name *filename*. The format of a config file is described within the documentation of the `ConfigParser` module of the Python Standard Library.

Each section of the config file becomes a data line. The options in a section are the columns. The name of the options will be used as file column names. All other parameters work as in *graph.data.file* and *graph.data.data* since they all use the same code.

4.5 Styles

Please note that we're talking about graph styles here. Those are responsible for plotting symbols, lines, bars and whatever else into a graph. Do not mix it up with path styles like the line width, the line style (solid, dashed, dotted *etc.*) and others.

The following classes provide styles to be used at the `plot()` method of a graph. The classes are implemented in `graph.style`.

class `symbolline`(*symbol=changecross, size="0.2 cm", errorscale=0.5, symbolattrs=[], errorbarattrs=[], lineattrs=[], epsilon=1e-10*)

This class is a style for plot symbols, lines and errorbars into a graph. *symbol* refers to a (changable) symbol method (see below). The symbol is drawn at size *size* (a visual PyX length; also changeable) using *symbolattrs*. *symbolattrs* are merged with the decorator `deco.stroked`. *errorscale* is the size of the error bars compared to the symbol size. *errorbarattrs* and *lineattrs* are strokestyles for stroking the errorbars and lines, respecively. *lineattrs* are merged with *changelinestyle* (see below). *epsilon* is used to determine, when a symbol is outside of the graph (in graph coordinates).

symbolline is useable on graphs with arbitrary dimension and geometry. It needs one data column for each graph dimension. The data names must be equal to an axis name. Furthermore there can be data names constructed out of the axis names for identifying data for the error bars. Suppose `X` is an axis name. Then `symbolline` allows for the following data names as well:

data name	**description**
`Xmin`	minimal value
`Xmax`	maximal value
`Xd`	minimal and maximal delta
`Xdmin`	minimal delta
`Xdmax`	maximal delta

Minimal and maximal values are calculated from delta by subtracting and adding it to the value itself. Most of the data names are mutal exclusive (whenever a minimal or maximal value would be set twice).

`symbolline` provides some symbol methods, namely:

`cross`(*x_pt, y_pt, size_pt*)

A cross. Should be used for stroking only.

`plus`(*x_pt, y_pt, size_pt*)

A plus. Should be used for stroking only.

`square`(*x_pt, y_pt, size_pt*)

A square. Might be stroked or filled or both.

`triangle`(*x_pt, y_pt, size_pt*)

A triangle. Might be stroked or filled or both.

`circle`(*x_pt, y_pt, size_pt*)

A circle. Might be stroked or filled or both.

`diamond`(*x_pt, y_pt, size_pt*)

A diamond. Might be stroked or filled or both.

`symbolline` provides some changeable symbol methods as class variables, namely:

`changecross`

attr.changelist([cross, plus, square, triangle, circle, diamond])

`changeplus`

attr.changelist([plus, square, triangle, circle, diamond, cross])

`changesquare`

attr.changelist([square, triangle, circle, diamond, cross, plus])

`changetriangle`

attr.changelist([triangle, circle, diamond, cross, plus, square])

`changecircle`

attr.changelist([circle, diamond, cross, plus, square, triangle])

`changediamond`

attr.changelist([diamond, cross, plus, square, triangle, circle])

changesquaretwice
attr.changelist([square, square, triangle, triangle, circle, circle, diamond, diamond])

changetriangletwice
attr.changelist([triangle, triangle, circle, circle, diamond, diamond, square, square])

changecircletwice
attr.changelist([circle, circle, diamond, diamond, square, square, triangle, triangle])

changediamondtwice
attr.changelist([diamond, diamond, square, square, triangle, triangle, circle, circle])

`symbolline` provides two changeable decorators for alternated filling and stroking. Those are especially usefull in combination with the `change-twice`-symbol methods above. They are:

changestrokedfilled
attr.changelist([deco.stroked, deco.filled])

changefilledstroked
attr.changelist([deco.filled, deco.stroked])

Finally, there is a changeable linestyle used by default. It is defined as:

changelinestyle
attr.changelist([style.linestyle.solid, style.linestyle.dashed, style.linestyle.dotted, style.linestyle.dashdotted])

class symbol(*symbol=changecross, size="0.2 cm", errorscale=0.5, symbolattrs=[], errorbarattrs=[], epsilon=1e-10*)
This class is a style to plot symbols and errorbars into a graph. It is equivalent to `symbollines` except that it does not allow for lines. An instance of `symbol` is the default style for all data classes described in section 4.4 except for `function` and `paramfunction`.

class line(*errorbarattrs=[]*)
This class is a style to stroke lines into graph. It is equivalent to `symbollines` except that it does not allow for symbols and errorbars. Thus it also does not accept data names for error bars. Instances of `line` are the default style for the data classes `function` and `paramfunction`.

class text(*textdx="0", textdy="0.3 cm", textattrs=[], symbol=changecross, size="0.2 cm", errorscale=0.5, symbolattrs=[], errorbarattrs=[], epsilon=1e-10*)
This class enhances `symbol` by adding text to the symbol. The text to be written has provided in the additional data column named `text`. *textdx* and *textdy* are the position of the text with respect of the symbol. *textattrs* are text attributes for the output of the text. All other parameters have the same meaning as in the `symbol` class.

class arrow(*linelength="0.25 cm", arrowsize="0.15 cm", lineattrs=[], arrowattrs=[], epsilon=1e-10*)
This class is a style to plot short lines with arrows into a two-dimensional graph. The position of the arrow is defined by two data columns named like an axes for each graph dimension. Two additional data columns named `size` and `angle` define the size and angle for each arrow. `size` is taken as a factor to *arrowsize* and *linelength*, the size of the arrow and the length of the line the arrow is plotted at. `angle` is the angle the arrow points to with respect to a horizonal lines. The `angle` is taken in degree and use in mathematical positive sense. *lineattrs* and *arrowattrs* are styles for the arrow line and arrow head respectively. *epsilon* is used to determine, when the arrow is outside of the graph (in graph coordinates).

class rect(*palette=color.palette.Gray*)
This class is a style to plot coloured rectangles into a two-dimensional graph. The position of the rectangles are given by 4 data columns named `Xmin` and `Xmax` where `X` stands for two axes names, one for each graph dimension. The additional data column named `color` specifies the color of the rectangle defined by *palette*. Thus the valid color range is [0:1].

Note: Although this style can be used for plotting coloured surfaces, it will lead to a huge memory footprint of PyXtogether with a long running time and large outputs. Improved support for coloured surfaces are planned for the future.

class bar(*fromvalue=None, frompathattrs=[], barattrs=[], subnames=None, epsilon=1e-10*)

This class is a style to plot bars into a two-dimensional graph. The bars are plotted on top of a specialized axis, namely a bar axis. The data column for this bar axis is named `Xname` where `X` is an axis name. The bar value have a name an axis of the other graph dimension. Suppose the name of this value axis is `Y` than you can stack further bars on top of this bar by providing additional data columns consecutively named `Ystack1`, `Ystack2`, `Ystack3`, *etc.* When plotting several bars in a single graph, those bars are placed side by side (at the same value of `Xname`). The name axis, a bar axis, must then be a nested bar axis. The names used for the subaxis can be set by *subnames*. When not set, integer numbers starting from zero will be used.

The bars start at *fromvalue* when provided. The *fromvalue* is marked by a gridline stroked using *frompathattrs*. The bars are filled using *barattrs*. *barattrs* is merged with '`[color.palette.Rainbow, deco.stroked([color.gray.black])]`' and iterated independently when several bars are plotted side by side and when several bars are plotted on top of each other. When mixing both possibilities, you may use nested changeable styles.

4.6 Keys

The following class provides a key, whose instances can be passed to the constructor keyword argument `key` of a graph. The class is implemented in `graph.key`.

class **`key`**(*dist="0.2 cm", pos="tr", hinside=1, vinside=1, hdist="0.6 cm", vdist="0.4 cm", symbolwidth="0.5 cm", symbolheight="0.25 cm", symbolspace="0.2 cm", textattrs=[]*)

This class writes the title of the data in a plot together with a small illustration of the style. The style is responsible for its illustration.

dist is a visual length and a distance between the key entries. *pos* is the position of the key with respect to the graph. Allowed values are combinations of '`t`' (top) and '`b`' (bottom) with '`l`' (left) and '`r`' (right). *hdist* and *vdist* are the distances from the specified corner of the graph. *hinside* and *vinside* are booleans to define whether the key should be placed horizontally and vertically inside of the graph or not.

symbolwidth and *symbolheight* is passed to the style to control the size of the style illustration. *symbolspace* is the space between the illustration and the text. *textattrs* are attributes for the text creation. They are merged with '`[text.vshift.mathaxis]`'.

CHAPTER

FIVE

Axes

Axes are a fundamental component of graphs although there might be use cases outside of the graph system. Internally axes are constructed out of components, which handle different tasks and axis need to fullfill:

axis
Basically a container for axis data and the components. It implements the conversion of a data value to a graph coordinate of range [0:1]. It does also handle the proper usage of the components in complicated tasks (*i.e.* combine the partitioner, texter, painter and rater to find the best partitioning).

tick
Ticks are plotted along the axis. They might be labeled with text as well.

partitioner, in the code the short form "parter" is used
Creates one or several choises of tick lists suitable to a certain axis range.

texter
Creates labels for ticks when they are not set manually.

painter
Responsible to paint the axis.

rater
Calculate ratings, which can be used to select the best suitable partitioning.

The names above map directly to modules, which are provided in the directory 'graph/axis'. Sometimes it might be conventient to import the axis directory directly rather access them through the graph. This would look like:

```
from pyx import *
graph.axis.painter() # and the like

from pyx.graph import axis
axis.painter() # this is shorter ...
```

In most cases different implementations are available through different classes, which can be combined in various ways. There are various axis examples distributed with PyX, where you can see some of the features of the axis with a few lines of code each. Hence we can here directly step on to the reference of the available components.

5.1 Axes

The following classes are part of the module `graph.axis.axis`. However, there is a shortcut to access those classes via `graph.axis` directly.

The position of an axis is defined by an instance of a class providing the following methods:

`basepath`(*x1=None, x2=None*)
Returns a path instance for the the base path. *x1* and *x2* are the axis range, the base path should cover.

vbasepath(*v1=None, v2=None*)
: Like `basepath` but in graph coordinates.

gridpath(*x*)
: Returns a path instance for the the grid path at position *x*. Might return `None` when no grid path is available.

vgridpath(*v*)
: Like `gridpath` but in graph coordinates.

tickpoint(*x*)
: Returns the position of *x* as a tuple '`(x, y)`'.

vtickpoint(*v*)
: Like `tickpoint` but in graph coordinates.

tickdirection(*x*)
: Returns the direction of a tick at *x* as a tuple '`(dx, dy)`'. The tick direction points inside of the graph.

vtickdirection(*v*)
: Like `tickdirection` but in graph coordinates.

Instances of the following classes can be passed to the ***axes* keyword arguments of a graph. Those instances should be used once only.

class linear(*min=None, max=None, reverse=0, divisor=None, title=None, parter=parter.autolinear(), manualticks=[], density=1, maxworse=2, rater=rater.linear(), texter=texter.mixed(), painter=painter.regular()*)
: This class provides a linear axis. *min* and *max* are the axis range. When not set, they are adjusted automatically by the data to be plotted in the graph. Note, that some data might want to access the range of an axis (*e.g.* the `function` class when no range was provided there) or you need to specify a range when using the axis without plugging it into a graph (*e.g.* when drawing a axis along a path).

  reverse can be set to indicate a reversed axis starting with bigger values first. Alternatively you can fix the axis range by *min* and *max* accordingly. When divisor is set, it is taken to divide all data range and position informations while creating ticks. You can create ticks not taking into account a factor by that. *title* is the title of the axis.

  parter is a partitioner instance, which creates suitable ticks for the axis range. Those ticks are merged with manual given ticks by *manualticks* before proceeding with rating, painting *etc.* Manually placed ticks win against those created by the partitioner. For automatic partitioners, which are able to calculate several possible tick lists for a given axis range, the *density* is a (linear) factor to favour more or less ticks. It should not be stressed to much (its likely, that the result would be unappropriate or not at all valid in terms of rating label distances). But within a range of say 0.5 to 2 (even bigger for large graphs) it can help to get less or more ticks than the default would lead to. *maxworse* is a the number of trials with more and less ticks when a better rating was already found. *rater* is a rater instance, which rates the ticks and the label distances for being best suitable. It also takes into account *density*. The rater is only needed, when the partitioner creates several tick lists.

  texter is a texter instance. It creates labels for those ticks, which claim to have a label, but do not have a label string set already. Ticks created by partitioners typically receive their label strings by texters. The *painter* is finally used to construct the output. Note, that usually several output constructions are needed, since the rater is also used to rate the distances between the label for an optimum.

class lin(*...*)
: This class is an abbreviation of `linear` described above.

class logarithmic(*min=None, max=None, reverse=0, divisor=None, title=None, parter=parter.autologarithmic(), manualticks=[], density=1, maxworse=2, rater=rater.logarithmic(), texter=texter.mixed(), painter=painter.regular()*)
: This class provides a logarithmic axis. All parameters work like `linear`. Only two parameters have a different default: *parter* and *rater*. Furthermore and most importantly, the mapping between data and graph coordinates is logarithmic.

class log(*...*)
: This class is an abbreviation of `logarithmic` described above.

class linked(*linkedaxis, painter=painter.linked()*)

This class provides an axis, which is linked to another axis instance. This means, it shares all its properties with the axis it is linked too except for the painter. Thus a linked axis is painted differently.

A standard use case are the `x2` and `y2` axes in an x-y-graph. Linked axes to the `x` and `y` axes are created automatically when not disabled by setting those axes to `None`. By that, ticks are stroked at both sides of an x-y-graph. However, linked axes can be used for in other cases as well. You can link axes within a graph or between different graphs as long as the orgininal axis is finished first (it must fix its layout first).

class `split`(*subaxes, splitlist=[0.5], splitdist=0.1, relsizesplitdist=1, title=None, painter=painter.split()*)

This class provides an axis, splitting the input values to its subaxes depeding on the range of the subaxes. Thus the subaxes need to have fixed range, up to the minimum of the first axis and the maximum of the last axis. *subaxes* actually takes the list of subaxes. *splitlist* defines the positions of the spliting in graph coordinates. Thus the length of *subaxes* must be the length of *splitlist* plus one. If an entry in *splitlist* is `None`, the axes aside define the split position taking into account the ratio of the axes ranges (meassured by an internal `relsize` attribute of each axis).

splitdist is the space reserved for a splitting in graph coordinates, when the corresponding entry in *splitlist* is not `None`. *relsizesplitdist* is the space reserved for the splitting in terms, when the corresponding entry in *splitlist* is `None` compared to the `relsize` of the axes aside.

title is the title of the split axes and *painter* is a specialized painter, which takes care of marking the axes breaks, while the painting of the subaxes are performed by their painters themself.

class `linkedsplit`(*linkedaxis, painter=painter.linkedsplit(), subaxispainter=omitsubaxispainter*)

This class provides an axis, which is linked to an instance of `split`. The purpose of a linked axis is described in class `linked` above. *painter* replaces the painter from the *linkedaxis* instance.

While this class creates linked axes for the subaxes of *linkedsplit* as well, the question arises what painters to use there. When *subaxispainter* is not set, no painter is given explicitly leaving this decision to the subaxes themself. This will lead to omitting all labels and the title. However, you can use a changeable attribute of painters in *subaxispainter* to replace the default.

class `bar`(*subaxis=None, multisubaxis=None, dist=0.5, firstdist=None, lastdist=None, title=None, painter=painter.bar()*)

This class provides an axis suitable for a bar style. It handles a discrete set of values and maps them to distinct ranges in graph coordinates. For that, the axis gets a list as data values. The first entry is taken to be one of the discrete values valid on this axis. All other parameters, lets call them others, are passed to a subaxis. When others has only one entry, it is passed as a value, otherwise as a list. The result of the conversion done by the subaxis is mapped into the graph coordinate range for this discrete value. When neigher *subaxis* nor *multisubaxis* is set, others must be a single value in range [0:1]. This value is used for the position at the subaxis without converion.

When *subaxis* is set, it is used for the conversion of others. When *multisubaxis* is set, it must be an instance of *bar* as well. It is than dublicated for each of the discrete values allowed for the axis. By that, you can create nested bar axes with a different discrete values for each discrete value of the axis. It is not allowed to set both, *subaxis* and *multisubaxis*.

dist is used as the spacing between the ranges for each distinct value. It is measured in the same units as the subaxis results, thus the default value of `0.5` means halve the width between the distinct values as the width for each distinct value. *firstdist* and *lastdist* are used before the first and after the last value. When set to `None`, halve of *dist* is used.

title is the title of the split axes and *painter* is a specialized painter for an bar axis. When *multisubaxis* is used, their painters are called as well, otherwise they are not taken into account.

`pathaxis`(*path, axis, direction=1*)

This function returns a (specialized) canvas containing the axis *axis* painted along the path *path*. *direction* defines the direction of the ticks. Allowed values are `1` (left) and `-1` (right).

5.2 Ticks

The following classes are part of the module `graph.axis.tick`.

class `rational`(*x, power=1, floatprecision=10*)

This class implements a rational number with infinite precision. For that it stores two integers, the enumerator `enum` and a denomintor `denom`. Note that the implementation of rational number arithmetics is not at all complete and designed for its special use case of axis parititioning in PyX preventing any roundoff errors.

x is the value of the rational created by a conversion from one of the following input values:

- A float. It is converted to a rational with finite precision determined by *floatprecision*.
- A string, which is parsed to a rational number with full precision. It is also allowed to provide a fraction like '`1/3`'.
- A sequence of two integers. Those integers are taken as enumerator and denominator of the rational.
- An instance defining instance variables `enum` and `denom` like `rational` itself.

power is an integer to calculate *x*`**`*power*. This is usefull at certain places in partitioners.

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

This class implements ticks based on rational numbers. Instances of this class can be passed to the `manualticks` parameter of a regular axis.

The parameters *x*, *power*, and *floatprecision* share its meaning with `rational`.

A tick has a tick level (*i.e.* markers at the axis path) and a label lavel (*e.i.* place text at the axis path), *ticklevel* and *labellevel*. These are non-negative integers or *None*. A value of `0` means a regular tick or label, `1` stands for a subtick or sublabel, `2` for subsubtick or subsublabel and so on. `None` means omitting the tick or label. *label* is the text of the label. When not set, it can be created automatically by a texter. *labelattrs* are the attributes for the labels.

5.3 Partitioners

The following classes are part of the module `graph.axis.parter`. Instances of the classes can be passed to the parter keyword argument of regular axes.

class `linear`(*tickdist=None, labeldist=None, extendtick=0, extendlabel=None, epsilon=1e-10*)

Instances of this class creates equally spaced tick lists. The distances between the ticks, subticks, subsubticks *etc.* starting from a tick at zero are given as first, second, third *etc.* item of the list *tickdist*. For a tick position, the lowest level wins, *i.e.* for `[2, 1]` even numbers will have ticks whereas subticks are placed at odd integer. The items of *tickdist* might be strings, floats or tuples as described for the *pos* parameter of class `tick`.

labeldist works equally for placing labels. When *labeldist* is kept `None`, labels will be placed at each tick position, but sublabels *etc.* will not be used. This copy behaviour is also available *vice versa* and can be disabled by an empty list.

extendtick can be set to a tick level for including the next tick of that level when the data exceed the range covered by the ticks by more then *epsilon*. *epsilon* is taken relative to the axis range. *extendtick* is disabled when set to `None` or for fixed range axes. *extendlabel* works similar to *extendtick* but for labels.

class `lin`(...)

This class is an abbreviation of `linear` described above.

class `autolinear`(*variants=defaultvariants, extendtick=0, epsilon=1e-10*)

Instances of this class creates equally spaced tick lists, where the distance between the ticks is adjusted to the range of the axis automatically. Variants are a list of possible choices for *tickdist* of `linear`. Further variants are build out of these by multiplying or dividing all the values by multiples of `10`. *variants* should be ordered that way, that the number of ticks for a given range will decrease, hence the distances between the ticks should increase within the *variants* list. *extendtick* and *epsilon* have the same meaning as in `linear`.

`defaultvariants`

```
[[tick.rational((1, 1)), tick.rational((1, 2))], [tick.rational((2,
1)), tick.rational((1, 1))], [tick.rational((5, 2)), tick.rational((5,
4))], [tick.rational((5, 1)), tick.rational((5, 2))]]
```

class `autolin`(...)

This class is an abbreviation of `autolinear` described above.

class `preexp`(*pres, exp*)

This is a storrage class defining positions of ticks on a logarithmic scale. It contains a list *pres* of positions pi and *exp*, a multiplicator m. Valid tick positions are defined by pim^n for any integer n.

class `logarithmic`(*tickpos=None, labelpos=None, extendtick=0, extendlabel=None, epsilon=1e-10*)

Instances of this class creates tick lists suitable to logarithmic axes. The positions of the ticks, subticks, subsubticks *etc.* are defined by the first, second, third *etc.* item of the list *tickpos*, which are all `preexp` instances.

labelpos works equally for placing labels. When *labelpos* is kept `None`, labels will be placed at each tick position, but sublabels *etc.* will not be used. This copy behaviour is also available *vice versa* and can be disabled by an empty list.

extendtick, *extendlabel* and *epsilon* have the same meaning as in `linear`.

Some `preexp` instances for the use in `logarithmic` are available as instance variables (should be used read-only):

`pre1exp5`

`preexp([tick.rational((1, 1))], 100000)`

`pre1exp4`

`preexp([tick.rational((1, 1))], 10000)`

`pre1exp3`

`preexp([tick.rational((1, 1))], 1000)`

`pre1exp2`

`preexp([tick.rational((1, 1))], 100)`

`pre1exp`

`preexp([tick.rational((1, 1))], 10)`

`pre125exp`

```
preexp([tick.rational((1, 1)), tick.rational((2, 1)), tick.rational((5,
1))], 10)
```

`pre1to9exp`

`preexp([tick.rational((1, 1)) for x in range(1, 10)], 10)`

class `log`(...)

This class is an abbreviation of `logarithmic` described above.

class `autologarithmic`(*variants=defaultvariants, extendtick=0, extendlabel=None, epsilon=1e-10*)

Instances of this class creates tick lists suitable to logarithmic axes, where the distance between the ticks is adjusted to the range of the axis automatically. Variants are a list of tuples with possible choices for *tickpos* and *labelpos* of `logarithmic`. *variants* should be ordered that way, that the number of ticks for a given range will decrease within the *variants* list.

extendtick, *extendlabel* and *epsilon* have the same meaning as in `linear`.

`defaultvariants`

```
[([log.pre1exp, log.pre1to9exp], [log.pre1exp, log.pre125exp]),
([log.pre1exp, log.pre1to9exp], None), ([log.pre1exp2, log.pre1exp],
None), ([log.pre1exp3, log.pre1exp], None), ([log.pre1exp4,
log.pre1exp], None), ([log.pre1exp5, log.pre1exp], None)]
```

class `autolog`(...)

This class is an abbreviation of `autologarithmic` described above.

5.4 Texter

The following classes are part of the module `graph.axis.texter`. Instances of the classes can be passed to the texter keyword argument of regular axes. Texters are used to define the label text for ticks, which request to have a label, but not label text was specified actually. A typical case are ticks created by partitioners described above.

class **`decimal`**(*prefix="", infix="", suffix="", equalprecision=0, decimalsep=".", thousandsep="", thousandthpartsep="", plus="", minus="-", period=r"\overline{%s}", labelattrs=[text.mathmode]*)

Instances of this class create decimal formatted labels.

The strings *prefix*, *infix*, and *suffix* are added to the label at the begin, immediately after the plus or minus, and at the end, respectively. *decimalsep*, *thousandsep*, and *thousandthpartsep* are strings used to separate integer from fractional part and three-digit groups in the integer and fractional part. The strings *plus* and *minus* are inserted in front of the unsigned value for non-negative and negative numbers, respectively.

The format string *period* should generate a period. It must contain one string insert operators '`%s`' for the period.

labelattrs is a list of attributes to be added to the label attributes given in the painter. It should be used to setup TeX features like `text.mathmode`. Text format options like `text.size` should instead be set at the painter.

class **`exponential`**(*plus="", minus="-", mantissaexp=r"{{%s}\cdot10^{%s}}", skipexp0=r"{%s}", skipexp1=None, nomantissaexp=r"{10^{%s}}", minusnomantissaexp=r"{-10^{%s}}", mantissamin=tick.rational((1, 1)), mantissamax=tick.rational((10L, 1)), skipmantissa1=0, skipallmantissa1=1, mantissatexter=decimal()*)

Instances of this class create decimal formatted labels with an exponential.

The strings *plus* and *minus* are inserted in front of the unsigned value of the exponent.

The format string *mantissaexp* should generate the exponent. It must contain two string insert operators '`%s`', the first for the mantissa and the second for the exponent. An alternative to the default is '`r"{{%s}{\rm e}{%s}}"`'.

The format string *skipexp0* is used to skip exponent `0` and must contain one string insert operator '`%s`' for the mantissa. `None` turns off the special handling of exponent `0`. The format string *skipexp1* is similar to *skipexp0*, but for exponent `1`.

The format string *nomantissaexp* is used to skip the mantissa `1` and must contain one string insert operator '`%s`' for the exponent. `None` turns off the special handling of mantissa `1`. The format string *minusnomantissaexp* is similar to *nomantissaexp*, but for mantissa `-1`.

The `tick.rational` instances *mantissamin*<*mantissamax* are minimum (including) and maximum (excluding) of the mantissa.

The boolean *skipmantissa1* enables the skipping of any mantissa equals `1` and `-1`, when *minusnomantissaexp* is set. When the boolean *skipallmantissa1* is set, a mantissa equals `1` is skipped only, when all mantissa values are `1`. Skipping of a mantissa is stronger than the skipping of an exponent.

mantissatexter is a texter instance for the mantissa.

class **`mixed`**(*smallestdecimal=tick.rational((1, 1000)), biggestdecimal=tick.rational((9999, 1)), equaldecision=1, decimal=decimal(), exponential=exponential()*)

Instances of this class create decimal formatted labels with an exponential, when the unsigned values are small or large compared to *1*.

The rational instances *smallestdecimal* and *biggestdecimal* are the smallest and biggest decimal values, where the decimal texter should be used. The sign of the value is ignored here. For a tick at zero the decimal texter is considered best as well. *equaldecision* is a boolean to indicate whether the decision for the decimal or exponential texter should be done globally for all ticks.

decimal and *exponential* are a decimal and an exponential texter instance, respectively.

class **`rational`**(*prefix="", infix="", suffix="", enumprefix="", enuminfix="", enumsuffix="", denomprefix="", denominfix="", denomsuffix="", plus="", minus="-", minuspos=0, over=r"%s\over%s", equaldenom=0, skip1=1, skipenum0=1, skipenum1=1, skipdenom1=1, labelattrs=[text.mathmode]*)

Instances of this class create labels formated as fractions.

The strings *prefix*, *infix*, and *suffix* are added to the label at the begin, immediately after the plus or minus, and at the end, respectively. The strings *prefixenum*, *infixenum*, and *suffixenum* are added to the labels enumerator accordingly whereas *prefixdenom*, *infixdenom*, and *suffixdenom* do the same for the denominator.

The strings *plus* and *minus* are inserted in front of the unsigned value. The position of the sign is defined by *minuspos* with values `1` (at the enumerator), `0` (in front of the fraction), and `-1` (at the denomerator).

The format string *over* should generate the fraction. It must contain two string insert operators '`%s`', the first for the enumerator and the second for the denominator. An alternative to the default is '`"{{%s}/{%s}}"`'.

Usually, the enumerator and denominator are canceled, while, when *equaldenom* is set, the least common multiple of all denominators is used.

The boolean *skip1* indicates, that only the prefix, plus or minus, the infix and the suffix should be printed, when the value is `1` or `-1` and at least one of *prefix*, *infix* and *suffix* is present.

The boolean *skipenum0* indicates, that only a `0` is printed when the enumerator is zero.

skipenum1 is like *skip1* but for the enumerator.

skipdenom1 skips the denominator, when it is `1` taking into account *denomprefix*, *denominfix*, *denomsuffix* *minuspos* and the sign of the number.

labelattrs has the same meaning than for *decimal*.

5.5 Painter

The following classes are part of the module `graph.axis.painter`. Instances of the painter classes can be passed to the painter keyword argument of regular axes.

class `rotatetext`(*direction, epsilon=1e-10*)

This helper class is used in direction arguments of the painters below to prevent axis labels and titles being written upside down. In those cases the text will be rotated by 180 degrees. *direction* is an angle to be used relative to the tick direction. *epsilon* is the value by which 90 degrees can be exceeded before an 180 degree rotation is performed.

The following two class variables are initialized with the most common use case:

`parallel`

`rotatetext(90)`

`orthogonal`

`rotatetext(180)`

class `ticklength`(*initial, factor*)

This helper class provides changeable PyX lengths starting from an initial value *initial* multiplied by *factor* again and again. The resulting lengths are thus a geometric series.

There are some class variables initialized with suitable values for tick stroking. They are named `ticklength.SHORT`, `ticklength.SHORt`, ..., `ticklength.short`, `ticklength.normal`, `ticklength.long`, ..., `ticklength.LONG`. `ticklength.normal` is initialized with a length of `0.12` and the reciprocal of the golden mean as `factor` whereas the others have a modified inital value by multiplication or division by multiples of $\sqrt{2}$ appropriately.

class `regular`(*innerticklength=ticklength.normal, outerticklength=None, tickattrs=[], gridattrs=None, basepathattrs=[], labeldist="0.3 cm", labelattrs=[], labeldirection=None, labelhequalize=0, labelvequalize=1, titledist="0.3 cm", titleattrs=[], titledirection=rotatetext.parallel, titlepos=0.5, texrunner=text.defaulttexrunner*)

Instances of this class are painters for regular axes like linear and logarithmic axes.

innerticklength and *outerticklength* are visual PyX lengths of the ticks, subticks, subsubticks *etc.* plotted along the axis inside and outside of the graph. Provide changeable attributes to modify the lengths of ticks compared to subticks *etc.* `None` turns off the ticks inside and outside the graph, respectively.

tickattrs and *gridattrs* are changeable stroke attributes for the ticks and the grid, where `None` turns off the feature. *basepathattrs* are stroke attributes for the axis or `None` to turn it off. *basepathattrs* is merged with '`[style.linecap.square]`'.

labeldist is the distance of the labels from the axis base path as a visual PyX length. *labelattrs* is a list of text attributes for the labels. It is merged with '`[text.halign.center, text.vshift.mathaxis]`'. *labeldirection* is an instance of *rotatetext* to rotate the labels relative to the axis tick direction or `None`.

The boolean values *labelhequalize* and *labelvequalize* force an equal alignment of all labels for straight vertical and horizontal axes, respectively.

titledist is the distance of the title from the rest of the axis as a visual PyX. *titleattrs* is a list of text attributes for the title. It is merged with '`[text.halign.center, text.vshift.mathaxis]`'. *titledirection* is an instance of *rotatetext* to rotate the title relative to the axis tick direction or `None`. *titlepos* is the position of the title in graph coordinates.

texrunner is the texrunner instance to create axis text like the axis title or labels.

class `linked`(*innerticklength=ticklength.short, outerticklength=None, tickattrs=[], gridattrs=None, basepathattrs=[], labeldist="0.3 cm", labelattrs=None, labeldirection=None, labelhequalize=0, labelvequalize=1, titledist="0.3 cm", titleattrs=None, titledirection=rotatetext.parallel, titlepos=0.5, texrunner=text.defaulttexrunner*)

This class is identical to `regular` up to the default values of *labelattrs* and *titleattrs*. By turning off those features, this painter is suitable for linked axes.

class `split`(*breaklinesdist="0.05 cm", breaklineslength="0.5 cm", breaklinesangle=-60, titledist="0.3 cm", titleattrs=None, titledirection=rotatetext.parallel, titlepos=0.5, texrunner=text.defaulttexrunner*)

Instances of this class are suitable painters for split axes.

breaklinesdist and *breaklineslength* are the distance between axes break markers in visual PyX lengths. *breaklinesangle* is the angle of the axis break marker with respect to the base path of the axis. All other parameters have the same meaning as in `regular`.

class `linkedsplit`(*breaklinesdist="0.05 cm", breaklineslength="0.5 cm", breaklinesangle=-60, titledist="0.3 cm", titleattrs=None, titledirection=rotatetext.parallel, titlepos=0.5, texrunner=text.defaulttexrunner*)

This class is identical to `split` up to the default value of *titleattrs*. By turning off this feature, this painter is suitable for linked split axes.

class `bar`(*innerticklength=None, outerticklength=None, tickattrs=[], basepathattrs=[], namedist="0.3 cm", nameattrs=[], namedirection=None, namepos=0.5, namehequalize=0, namevequalize=1, titledist="0.3 cm", titleattrs=[], titledirection=rotatetext.parallel, titlepos=0.5, texrunner=text.defaulttexrunner*)

Instances of this class are suitable painters for bar axes.

innerticklength and *outerticklength* are visual PyX length to mark the different bar regions along the axis inside and outside of the graph. `None` turns off the ticks inside and outside the graph, respectively. *tickattrs* are stroke attributes for the ticks or `None` to turns all ticks off.

The parameters with prefix *name* are identical to their *label* counterparts in `regular`. All other parameters have the same meaning as in `regular`.

class `linkedbar`(*innerticklength=None, outerticklength=None, tickattrs=[], basepathattrs=[], namedist="0.3 cm", nameattrs=None, namedirection=None, namepos=0.5, namehequalize=0, namevequalize=1, titledist="0.3 cm", titleattrs=None, titledirection=rotatetext.parallel, titlepos=0.5, texrunner=text.defaulttexrunner*)

This class is identical to `bar` up to the default values of *nameattrs* and *titleattrs*. By turning off those features, this painter is suitable for linked bar axes.

5.6 Rater

The rating of axes is implemented in `graph.axis.rater`. When an axis partitioning scheme returns several partitioning possibilities, the partitions needs to be rated by a positive number. The lowest rated axis partitioning is considered best.

The rating consists of two steps. The first takes into account only the number of ticks, subticks, labels and so on in comparison to optimal numbers. Additionally, the extension of the axis range by ticks and labels is taken into account. This rating leads to a preselection of possible partitions. In the second step, after the layout of prefered partitionings is calculated, the distance of the labels in a partition is taken into account as well at a smaller weight factor by default. Thereby partitions with overlapping labels will be rejected completely. Exceptionally sparse or dense labels will receive a bad rating as well.

class `cube`(*opt, left=None, right=None, weight=1*)

Instances of this class provide a number rater. *opt* is the optimal value. When not provided, *left* is set to `0` and *right* is set to `3*`*opt*. Weight is a multiplicator to the result.

The rater calculates *widht*`*((x-`*opt*`)/(other-`*opt*`))**3` to rate the value `x`, where `other` is *left* (`x`<*opt*) or *right* (`x`>*opt*).

class `distance`(*opt, weight=0.1*)

Instances of this class provide a rater for a list of numbers. The purpose is to rate the distance between label boxes. *opt* is the optimal value.

The rater calculates the sum of *weight*`*(`*opt*`/x-1)` (`x`<*opt*) or *weight*`*(x/`*opt*`-1)` (`x`>*opt*) for all elements `x` of the list. It returns this value divided by the number of elements in the list.

class `rater`(*ticks, labels, range, distance*)

Instances of this class are raters for axes partitionings.

ticks and *labels* are both lists of number rater instances, where the first items are used for the number of ticks and labels, the second items are used for the number of subticks (including the ticks) and sublabels (including the labels) and so on until the end of the list is reached or no corresponding ticks are available.

range is a number rater instance which rates the range of the ticks relative to the range of the data.

distance is an distance rater instance.

class `linear`(*ticks=[cube(4), cube(10, weight=0.5)], labels=[cube(4)], range=cube(1, weight=2), distance=distance("1 cm")*)

This class is suitable to rate partitionings of linear axes. It is equal to `rater` but defines predefined values for the arguments.

class `lin`(*...*)

This class is an abbreviation of `linear` described above.

class `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), distance=distance("1 cm")*)

This class is suitable to rate partitionings of logarithmic axes. It is equal to `rater` but defines predefined values for the arguments.

class `log`(*...*)

This class is an abbreviation of `logarithmic` described above.

CHAPTER

SIX

Module box: convex box handling

This module has a quite internal character, but might still be useful from the users point of view. It might also get further enhanced to cover a broader range of standard arranging problems.

In the context of this module a box is a convex polygon having optionally a center coordinate, which plays an important role for the box alignment. The center might not at all be central, but it should be within the box. The convexity is necessary in order to keep the problems to be solved by this module quite a bit easier and unambiguous.

Directions (for the alignment etc.) are usually provided as pairs (dx, dy) within this module. It is required, that at least one of these two numbers is unequal to zero. No further assumptions are taken.

6.1 polygon

A polygon is the most general case of a box. It is an instance of the class `polygon`. The constructor takes a list of points (which are (x, y) tuples) in the keyword argument `corners` and optionally another (x, y) tuple as the keyword argument `center`. The corners have to be ordered counterclockwise. In the following list some methods of this `polygon` class are explained:

`path(centerradius=None, bezierradius=None, beziersoftness=1):` returns a path of the box; the center might be marked by a small circle of radius `centerradius`; the corners might be rounded using the parameters `bezierradius` and `beziersoftness`. For each corner of the box there may be one value for beziersoftness and two bezierradii. For convenience, it is not necessary to specify the whole list (for beziersoftness) and the whole list of lists (bezierradius) here. You may give a single value and/or a 2-tuple instead.

`transform(*trafos):` performs a list of transformations to the box

`reltransform(*trafos):` performs a list of transformations to the box relative to the box center

`circlealignvector(a, dx, dy):` returns a vector (a tuple (x, y)) to align the box at a circle with radius `a` in the direction (`dx`, `dy`); see figure 6.1

`linealignvector(a, dx, dy):` as above, but align at a line with distance `a`

`circlealign(a, dx, dy):` as circlealignvector, but perform the alignment instead of returning the vector

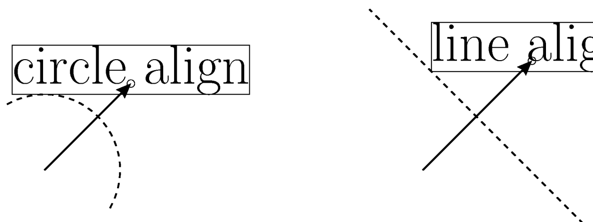


Figure 6.1: circle and line alignment examples (equal direction and distance)

`linealign(a, dx, dy):` as linealignvector, but perform the alignment instead of returning the vector

`extent(dx, dy):` extent of the box in the direction (`dx`, `dy`)

`pointdistance(x, y):` distance of the point (`x`, `y`) to the box; the point must be outside of the box

`boxdistance(other):` distance of the box to the box `other`; when the boxes are overlapping, `BoxCrossError` is raised

`bbox():` returns a bounding box instance appropriate to the box

6.2 functions working on a box list

`circlealignequal(boxes, a, dx, dy):` Performs a circle alignment of the boxes `boxes` using the parameters `a`, `dx`, and `dy` as in the `circlealign` method. For the length of the alignment vector its largest value is taken for all cases.

`linealignequal(boxes, a, dx, dy):` as above, but performing a line alignment

`tile(boxes, a, dx, dy):` tiles the boxes `boxes` with a distance `a` between the boxes (additional the maximal box extent in the given direction (`dx`, `dy`) is taken into account)

6.3 rectangular boxes

For easier creation of rectangular boxes, the module provides the specialized class `rect`. Its constructor first takes four parameters, namely the x, y position and the box width and height. Additionally, for the definition of the position of the center, two keyword arguments are available. The parameter `relcenter` takes a tuple containing a relative x, y position of the center (they are relative to the box extent, thus values between `0` and `1` should be used). The parameter `abscenter` takes a tuple containing the x and y position of the center. This values are measured with respect to the lower left corner of the box. By default, the center of the rectangular box is set to this lower left corner.

CHAPTER

SEVEN

Module connector

This module provides classes for connecting two `box`-instances with lines, arcs or curves. All constructors of the following connector-classes take two `box`-instances as first arguments. They return a `normpath`-instance from the first to the second box, starting/ending at the boxes' outline `path`. The behaviour of the path is determined by the boxes' center and some angle- and distance-keywords. The resulting path will additionally be shortened by lengths given in the `boxdists`-keyword (a list of two lengths, default `[0,0]`).

7.1 Class line

The constructor of the `line` class accepts only boxes and the `boxdists`-keyword.

7.2 Class arc

The constructor also takes either the `relangle`-keyword or a combination of `relbulge` and `absbulge`. The "bulge" is the greatest distance between the connecting arc and the straight connecting line. (Default: `relangle=45`, `relbulge=None`, `absbulge=None`)

Note that the bulge- override the angle-keyword. When both `relbulge` and `absbulge` are given they will be added.

7.3 Class curve

The construktor takes both angle- and bulge-keywords. Here, the bulges are used as distances between bezier-curve control points:

`absangle1` or `relangle1`
`absangle2` or `relangle2`, where the absolute angle overrides the relative if both are given. (Default: `relangle1=45`, `relangle2=45`, `absangle1=None`, `absangle2=None`)

`absbulge` and `relbulge`, where they will be added if both are given.
(Default: `absbulge`=None `relbulge`=0.39; these default values produce similar output like the defaults of the arc-class.)

Note that relative angle-keywords are counted in the following way: `relangle1` is counted in negative direction, starting at the straight connector line, and `relangle2` is counted in positive direction. Therefore, the outcome with two positive relative angles will always leave the straight connector at its left and will not cross it.

7.4 Class twolines

This class returns two connected straight lines. There is a vast variety of combinations for angle- and length-keywords. The user has to make sure to provide a non-ambiguous set of keywords:

`absangle1` or `relangle1` for the first angle,
`relangleM` for the middle angle and
`absangle2` or `relangle2` for the ending angle. Again, the absolute angle overrides the relative if both are given. (Default: all five angles are `None`)

`length1` and `length2` for the lengths of the connecting lines. (Default: `None`)

CHAPTER

EIGHT

Module epsfile: EPS file inclusion

With help of the `epsfile.epsfile` class, you can easily embed another EPS file in your canvas, thereby scaling, aligning the content at discretion. The most simple example looks like

```
from pyx import *
c = canvas.canvas()
c.insert(epsfile.epsfile(0, 0, "file.eps"))
c.writeEPSfile("output")
```

All relevant parameters are passed to the `epsfile.epsfile` constructor. They are summarized in the following table:

argument name	description
`x`	x-coordinate of position (measured in user units by default).
`y`	y-coordinate of position (measured in user units by default).
`filename`	Name of the EPS file (including a possible extension).
`width=None`	Desired width of EPS graphics or `None` for original width. Cannot be combined with scale specification.
`heigth=None`	Desired height of EPS graphics or `None` for original height. Cannot be combined with scale specification.
`scale=None`	Scaling factor for EPS graphics or `None` for no scaling. Cannot be combined with width or height specification.
`align="bl"`	Alignment of EPS graphics. The first character specifies the vertical alignment: `b` for bottom, `c` for center, and `t` for top. The second character fixes the horizontal alignment: `l` for left, `c` for center `r` for right.
`clip=1`	Clip to bounding box of EPS file?
`translatebbox=1`	Use lower left corner of bounding box of EPS file? Set to 0 with care.
`bbox=None`	If given, use `bbox` instance instead of bounding box of EPS file.

CHAPTER

NINE

Module bbox

The `bbox` module contains the definition of the `bbox` class representing bounding boxes of graphical elements like paths, canvases, etc. used in PyX. Usually, you obtain `bbox` instances as return values of the corresponding `bbox())` method, but you may also construct a bounding box by yourself.

9.1 bbox constructor

The `bbox` constructor accepts the following keyword arguments

keyword	description
`llx`	`None` (default) for $-\infty$ or x-position of the lower left corner of the bbox (in user units)
`lly`	`None` (default) for $-\infty$ or y-position of the lower left corner of the bbox (in user units)
`urx`	`None` (default) for ∞ or x-position of the upper right corner of the bbox (in user units)
`ury`	`None` (default) for ∞ or y-position of the upper right corner of the bbox (in user units)

9.2 bbox methods

`bbox` method	function
`intersects(other)`	returns `1` if the `bbox` instance and `other` intersect with each other.
`transformed(self, trafo)`	returns `self` transformed by transformation `trafo`.
`enlarged(all=0, bottom=None, left=None, top=None, right=None)`	return the bounding box enlarged by the given amount (in visual units). `all` is the default for all other directions, which is used whenever `None` is given for the corresponding direction.
`path()` or `rect()`	return the `path` corresponding to the bounding box rectangle.
`height()`	returns the height of the bounding box (in PyX lengths).
`width()`	returns the width of the bounding box (in PyX lengths).
`top()`	returns the y-position of the top of the bounding box (in PyX lengths).
`bottom()`	returns the y-position of the bottom of the bounding box (in PyX lengths).
`left()`	returns the x-position of the left side of the bounding box (in PyX lengths).
`right()`	returns the x-position of the right side of the bounding box (in PyX lengths).

Furthermore, two bounding boxes can be added (giving the bounding box enclosing both) and multiplied (giving the intersection of both bounding boxes).

CHAPTER

TEN

Module color

10.1 Color models

PostScript provides different color models. They are available to PyX by different color classes, which just pass the colors down to the PostScript level. This implies, that there are no conversion routines between different color models available. However, some color model conversion routines are included in Python's standard library in the module `colorsym`. Furthermore also the comparision of colors within a color model is not supported, but might be added in future versions at least for checking color identity and for ordering gray colors.

There is a class for each of the supported color models, namely `gray`, `rgb`, `cmyk`, and `hsb`. The constructors take variables appropriate to the color model. Additionally, a list of named colors is given in appendix B.

10.2 Example

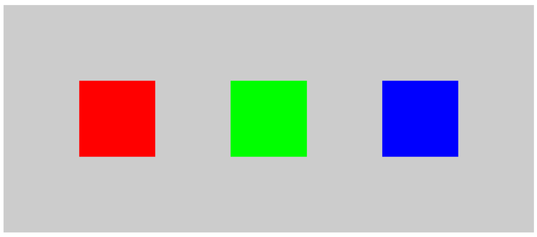
```
from pyx import *

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 file `color.eps` is created and looks like:



10.3 Color palettes

The color module provides a class `palette`. The constructor of that class receives two colors from the same color model and two named parameters `min` and `max`, which are set to `0` and `1` by default. Between those colors a linear interpolation takes place by the method `getcolor` depending on a value between `min` and `max`.

A list of named palettes is available in appendix C.

CHAPTER

ELEVEN

Module unit

With the `unit` module PyX makes available classes and functions for the specification and manipulation of lengths. As usual, lengths consist of a number together with a measurement unit, e.g., 1 cm, 50 points, 0.42 inch. In addition, lengths in PyX are composed of the five types “true”, “user”, “visual”, “width”, and “TEX”, e.g., 1 user cm, 50 true points, $(0.42\ \mathrm{visual} + 0.2\ \mathrm{width})$ inch. As their names indicate, they serve different purposes. True lengths are not scalable and are mainly used for return values of PyX functions. The other length types can be rescaled by the user and differ with respect to the type of object they are applied to:

user length: used for lengths of graphical objects like positions etc.

visual length: used for sizes of visual elements, like arrows, graph symbols, axis ticks, etc.

width length: used for line widths

TEX length: used for all TEX and LATEX output

For instance, if you only want thicker lines for a publication version of your figure, you can just rescale the width lengths. How this all works, is described in the following sections.

11.1 Class length

The constructor of the `length` class accepts as first argument either a number or a string:

- `length(number)` means a user length in units of the default unit, defined via `unit.set(defaultunit=defaultunit)`.
- For `length(string)`, the `string` has to consist of a maximum of three parts separated by one or more whitespaces:

  quantifier: integer/float value. Optional, defaults to `1`.

  type: `"t"` (true), `"u"` (user), `"v"` (visual), `"w"` (width), or `"x"` (TeX). Optional, defaults to `"u"`.

  unit: `"m"`, `"cm"`, `"mm"`, `"inch"`, or `"pt"`. Optional, defaults to the default unit.

The default for the first argument is chosen in such a way that `5*length()==length(5)`. Note that the default unit is initially set to `"cm"`, but can be changed at any time by the user. For instance, use

```
unit.set(defaultunit="inch")
```

if you want to specify per default every length in inches. Furthermore, the scaling of the user, visual and width types can be changed with the `set` function, as well. To this end, `set` accepts the named arguments `uscale`, `vscale`, and `wscale`. For example, if you like to change the thickness of all lines (with predefined linewidths) by a factor of two, just insert

```
unit.set(wscale = 2)
```

at the beginning of your program.

To complete the discussion of the `length` class, we mention, that as expected PyX lengths can be added, subtracted, multiplied by a numerical factor, converted to a string and compared with each other.

11.2 Subclasses of length

A number of subclasses of `length` are already predefined. They only differ in their defaults for `type` and `unit`. Note that again the default value for the quantifier is `1`, such that, for instance, `5*m(1)==m(5)`.

Subclass of `length`	Type	Unit	Subclass of `length`	Type	Unit
`m(x)`	user	m	`v_m(x)`	visual	m
`cm(x)`	user	cm	`v_cm(x)`	visual	cm
`mm(x)`	user	mm	`v_mm(x)`	visual	mm
`inch(x)`	user	inch	`v_inch(x)`	visual	inch
`pt(x)`	user	points	`v_pt(x)`	visual	points
`t_m(x)`	true	m	`w_m(x)`	width	m
`t_cm(x)`	true	cm	`w_cm(x)`	width	cm
`t_mm(x)`	true	mm	`w_mm(x)`	width	mm
`t_inch(x)`	true	inch	`w_inch(x)`	width	inch
`t_pt(x)`	true	points	`w_pt(x)`	width	points
`u_m(x)`	user	m	`x_m(x)`	TeX	m
`u_cm(x)`	user	cm	`x_cm(x)`	TeX	cm
`u_mm(x)`	user	mm	`x_mm(x)`	TeX	mm w
`u_inch(x)`	user	inch	`x_inch(x)`	TeX	inch
`u_pt(x)`	user	points	`x_pt(x)`	TeX	points

Here, `x` is either a number or a string, which, as mentioned above, defaults to `1`.

11.3 Conversion functions

If you want to know the value of a PyX length in certain units, you may use the predefined conversion functions which are given in the following table

function	result
`to_m(l)`	`l` in units of m
`to_cm(l)`	`l` in units of cm
`to_mm(l)`	`l` in units of mm
`to_inch(l)`	`l` in units of inch
`to_pt(l)`	`l` in units of points

If `l` is not yet a `length` instance, it is converted first into one, as described above. You can also specify a tuple, if you want to convert multiple lengths at once.

CHAPTER

TWELVE

Module trafo: linear transformations

With the `trafo` modulo PyX supports linear transformations, which can then be applied to canvases, Bézier paths and other objects. It consists of the main class `trafo` representing a general linear transformation and subclasses thereof, which provide special operations like translation, rotation, scaling, and mirroring.

12.1 Class trafo

The `trafo` class represents a general linear transformation, which is defined for a vector $\vec{x}$ as

$$\vec{x}' = \mathsf{A}\,\vec{x} + \vec{b}\,,$$

where A is the transformation matrix and $\vec{b}$ the translation vector. The transformation matrix must not be singular, *i.e.* we require $\det \mathsf{A} \neq 0$.

Multiple `trafo` instances can be multiplied, corresponding to a consecutive application of the respective transformation. Note that `trafo1*trafo2` means that `trafo1` is applied after `trafo2`, *i.e.* the new transformation is given by $\mathsf{A} = \mathsf{A}1\mathsf{A}2$ and $\vec{b} = \mathsf{A}1\vec{b}2 + \vec{b}1$. Use the `trafo` methods described below, if you prefer thinking the other way round. The inverse of a transformation can be obtained via the `trafo` method `inverse()`, defined by the inverse A^{-1} of the transformation matrix and the translation vector $-\mathsf{A}^{-1}\vec{b}$.

The methods of the `trafo` class are summarized in the following table.

`trafo` method	function
`__init__(matrix=((1,0),(0,1)), vector=(0,0)):`	create new `trafo` instance with transformation `matrix` and `vector`.
`apply(x, y)`	apply `trafo` to point vector $(\mathtt{x}, \mathtt{y})$.
`inverse()`	returns inverse transformation of `trafo`.
`mirrored(angle)`	returns `trafo` followed by mirroring at line through $(0,0)$ with direction `angle` in degrees.
`rotated(angle, x=None, y=None)`	returns `trafo` followed by rotation by `angle` degrees around point $(\mathtt{x}, \mathtt{y})$, or $(0,0)$, if not given.
`scaled(sx, sy=None, x=None, y=None)`	returns `trafo` followed by scaling with scaling factor `sx` in x-direction, `sy` in y-direction ($\mathtt{sy} = \mathtt{sx}$, if not given) with scaling center $(\mathtt{x}, \mathtt{y})$, or $(0,0)$, if not given.
`translated(x, y)`	returns `trafo` followed by translation by vector $(\mathtt{x}, \mathtt{y})$.
`slanted(a, angle=0, x=None, y=None)`	returns `trafo` followed by XXX

12.2 Subclasses of trafo

The `trafo` module provides provides a number of subclasses of the `trafo` class, each of which corresponds to one `trafo` method. They are listed in the following table:

`trafo` subclass	function
`mirror(angle)`	mirroring at line through $(0,0)$ with direction `angle` in degrees.
`rotate(angle, x=None, y=None)`	rotation by `angle` degrees around point $(\mathtt{x},\mathtt{y})$, or $(0,0)$, if not given.
`scale(sx, sy=None, x=None, y=None)`	scaling with scaling factor `sx` in x-direction, `sy` in y-direction ($\mathtt{sy} = \mathtt{sx}$, if not given) with scaling center $(\mathtt{x},\mathtt{y})$, or $(0,0)$, if not given.
`translate(x, y)`	translation by vector $(\mathtt{x},\mathtt{y})$.
`slant(a, angle=0, x=None, y=None)`	XXX

APPENDIX

A

Mathematical expressions

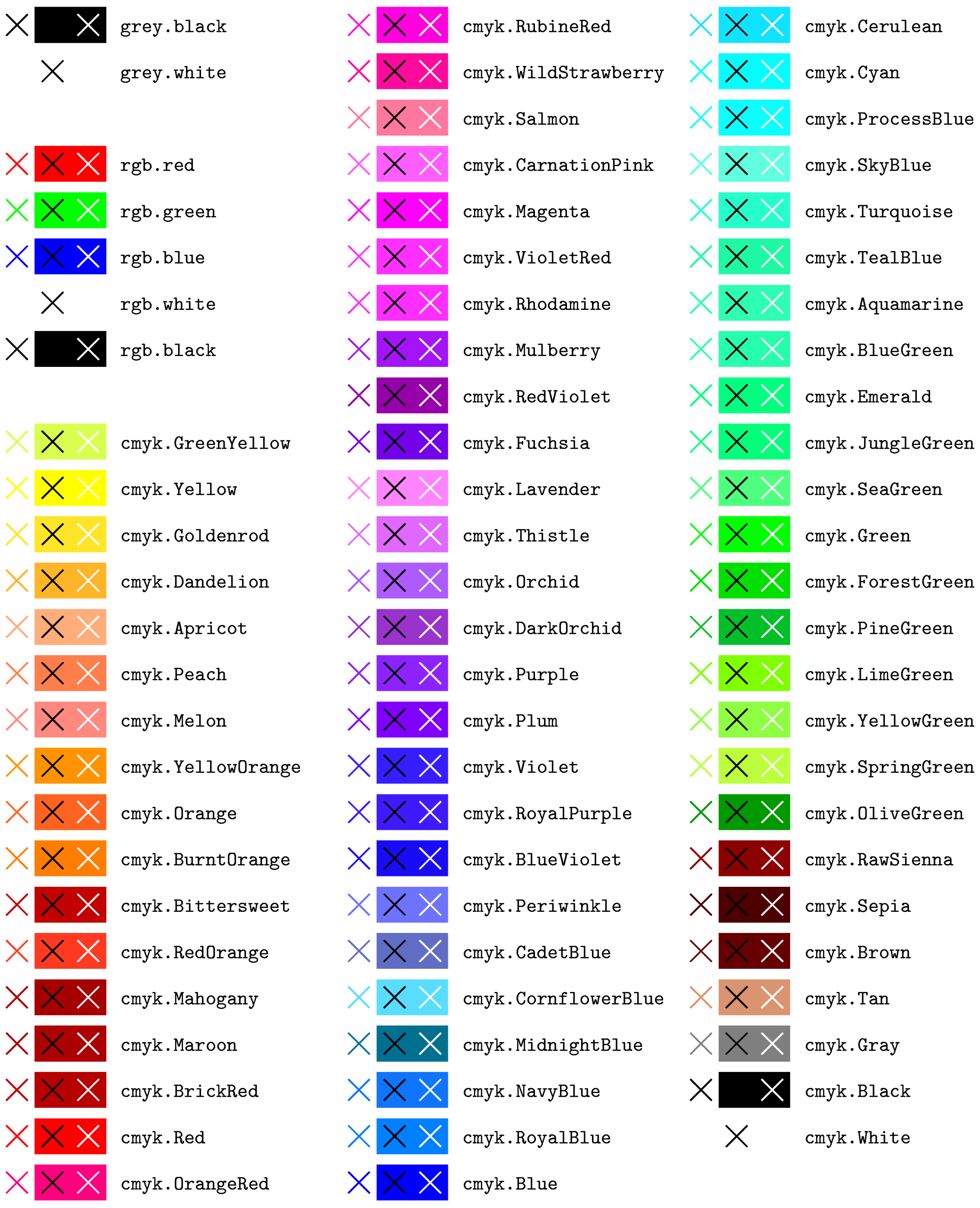
At several points within PyX mathematical expressions can be provided in form of string parameters. They are evaluated by the module `mathtree`. This module is not described futher in this user manual, because it is considered to be a technical detail. We just give a list of available operators, functions and predefined variable names here here.

Operators: `+`; `-`; `*`; `/`; `**`

Functions: `neg` (negate); `abs` (absolute value); `sgn` (signum); `sqrt` (square root); `exp`; `log` (natural logarithm); `sin`, `cos`, `tan`, `asin`, `acos`, `atan` (trigonometric functions in radian units); `sind`, `cosd`, `tand`, `asind`, `acosd`, `atand` (as before but in degree units); `norm` ($\sqrt{a^2 + b^2}$ as an example for functions with multiple arguments)

predefined variables: `pi` (π); `e` (e)

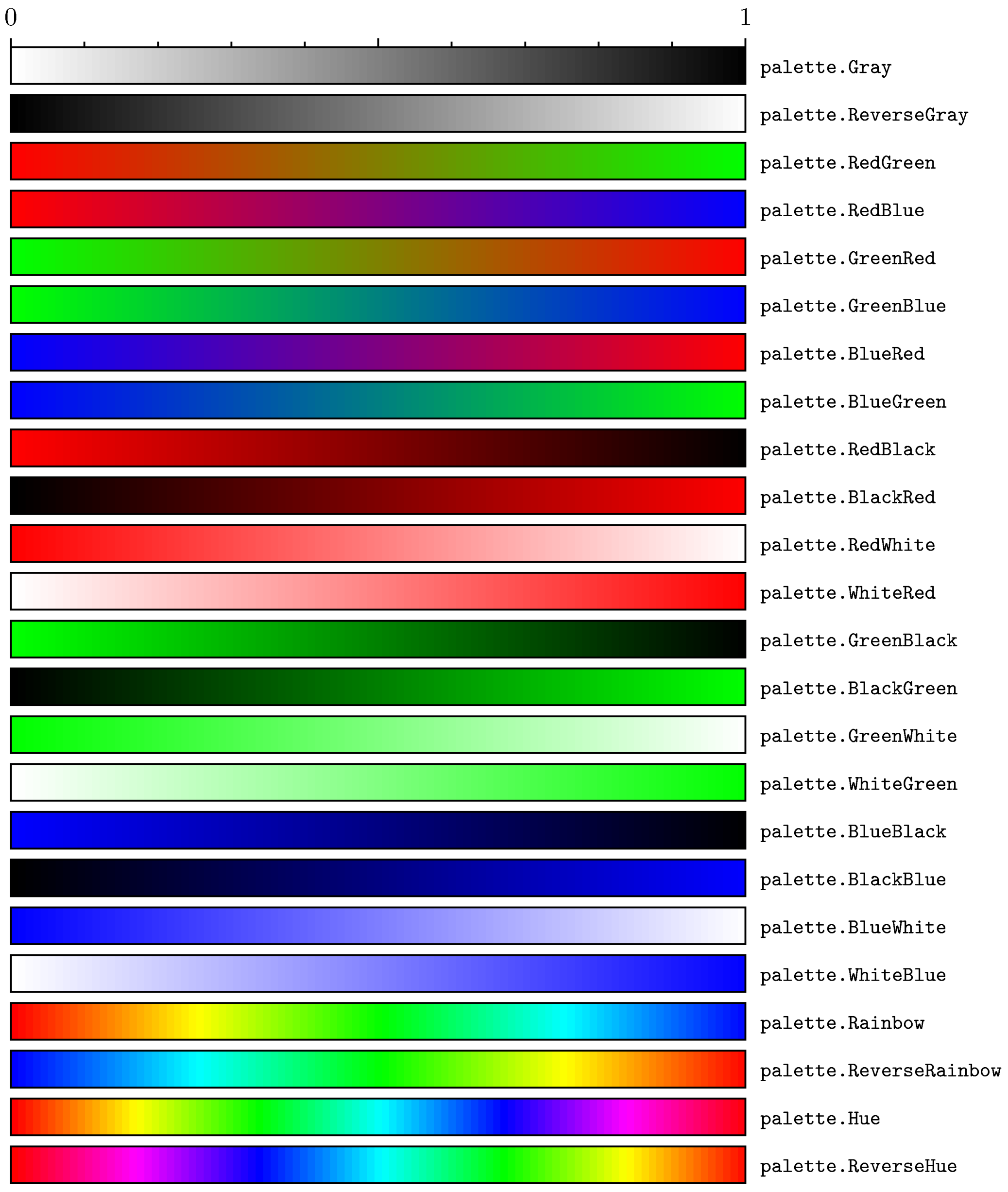
Named colors



APPENDIX

C

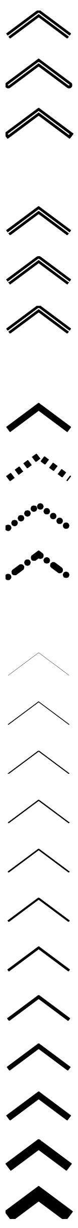
Named palettes



APPENDIX

D

Module `style`

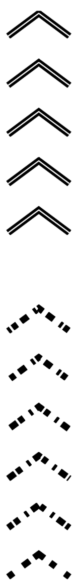


`linecap.butt` (default)
`linecap.round`
`linecap.square`

`linejoin.miter` (default)
`linejoin.round`
`linejoin.bevel`

`linestyle.solid` (default)
`linestyle.dashed`
`linestyle.dotted`
`linestyle.dashdotted`

`linewidth.THIN`
`linewidth.THIn`
`linewidth.THin`
`linewidth.Thin`
`linewidth.thin`
`linewidth.normal` (default)
`linewidth.thick`
`linewidth.Thick`
`linewidth.THick`
`linewidth.THIck`
`linewidth.THICk`
`linewidth.THICK`



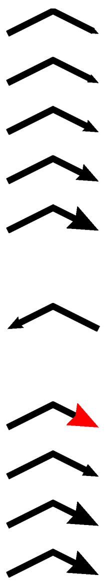
`miterlimit.lessthan180deg`
`miterlimit.lessthan90deg`
`miterlimit.lessthan60deg`
`miterlimit.lessthan45deg`
`miterlimit.lessthan11deg` (default)

`dash((1, 1, 2, 2, 3, 3), 0)`
`dash((1, 1, 2, 2, 3, 3), 1)`
`dash((1, 2, 3), 2)`
`dash((1, 2, 3), 3)`
`dash((1, 2, 3), 4)`
`dash((1, 2, 3), rellengths=1)`

APPENDIX

E

Arrows in deco module



```
earrow.Small
earrow.small
earrow.normal
earrow.large
earrow.Large

barrow.normal

earrow.Large([deco.filled([color.rgb.red]), style.linewidth.normal])
earrow.normal(constriction=0)
earrow.Large([style.linejoin.round])
earrow.Large([deco.stroked.clear])
```

INDEX