http://pyx.sourceforge.net/

PyX 0.4.1
User Manual

Jörg Lehmann <joergl@users.sourceforge.net>
André Wobst <wobsta@users.sourceforge.net>

September 23, 2003

PostScript is a trademark of Adobe Systems Incorporated.

Contents

1. Introduction

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.

2. 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 four types "true", "user", "visual" and "width", *e.g.* 1 user cm, 50 true points, (0.42 visual + 0.2 width) inch. As their names indicate, they serve different purposes. True lengths are not scalable and serve mainly for return values of PyX functions. The other length types allow a rescaling 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

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.

2.1. 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), or `"w"` (width). 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 and converted to a string.

2.2. 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			
`u_cm(x)`	user	cm			
`u_mm(x)`	user	mm			
`u_inch(x)`	user	inch			
`u_pt(x)`	user	points			

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

2.3. 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.

3. Module path: PostScript like paths

With the help of the path module it is possible to construct PostScript like paths, which are one of the main building blocks for the generation of drawings. To that end it provides

- classes (derived from `pathel`) for the primitives `moveto`, `lineto`, etc.
- the class `path` (and derivatives thereof) representing an entire PostScript path
- the class `normpath` (and derivatives thereof) which is a path consisting only of a certain subset of `pathel`s, namely the four `normpathel`s `moveto`, `lineto`, `curveto` and `closepath`.

3.1. Class pathel

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.

Subclass of `pathel`	function
`closepath()`	closes current subpath
`moveto(x, y)`	sets current point to (`x`, `y`)
`rmoveto(dx, dy)`	moves current point by (`dx`, `dy`)
`lineto(x, y)`	moves current point to (`x`, `y`) while drawing a straight line
`rlineto(dx, dy)`	moves current point by by (`dx`, `dy`) while drawing a straight line
`arc(x, y, r, angle1, angle2)`	appends arc segment in counterclockwise direction with center (`x`, `y`) and radius `r` from `angle1` to `angle2` (in degrees).
`arcn(x, y, r, angle1, angle2)`	appends arc segment in clockwise direction with center (`x`, `y`) and radius `r` from `angle1` to `angle2` (in degrees).
`arct(x1, y1, x2, y2, r)`	appends arc segment of radius `r` connecting between (`x1`, `y1`) and (`x2`, `y2`).
`rcurveto(dx1, dy1, dx2, dy2, dx3, dy3)`	appends a Bézier curve with the following four control points: current point and the points defined relative to the current point by (`dx1`, `dy1`), (`dx2`, `dy2`), and (`dx3`, `dy3`)

Some notes on the above:

- All coordinates are in PyX lengths
- If the current point is defined before an `arc` or `arcn` command, a straight line from current point to the beginning of the arc is prepended.
- The bounding box (see below) of Bézier curves is actually the box enclosing the control points, *i.e.* not neccesarily the smallest rectangle enclosing the Bézier curve.

3.2. Class path

The class path represents PostScript like paths in PyX. The `path` constructor allows the creation of such a path out of a series of `pathel`s. The following simple example generates a triangle: looks like:

```
from pyx import *
from pyx.path import *

p = path(moveto(0, 0),
         lineto(0, 1),
         lineto(1, 1),
         closepath())
```

In section 5, we shall see, how it is possible to output such a path on a canvas. For the moment, we only want to discuss the methods provided by the `path` class. These range from standard operations like the determination of the length of a path via `len(p)`, fetching of items using `p[index]` and the possibility to concatenate two paths, `p1 + p2`, append further `pathel`s using `p.append(pathel)` to more advanced methods, which are summarized in the following table.
XXX terminology: subpath, ...

path method	function
`__init__(*pathels)`	construct new `path` consisting of `pathels`
`append(pathel)`	appends `pathel` to the end of `path`
`arclength(epsilon=1e-5)`	returns the total arc length of all `path` segments in PostScript points with accuracy `epsilon`.†
`at(t)`	returns the coordinates of the point of `path` corresponding to the parameter value `t`.†
`lentopar(l, epsilon=1e-5)`	returns the parameter value corresponding to the lengths `l` (one or a list of lengths). This uses arclength-calculations with accuracy `epsilon`.†
`bbox()`	returns the bounding box of the `path`
`begin()`	return first point of first subpath of `path`.†
`end()`	return last point of last subpath of `path`.†
`glue(opath)`	returns the `path` glued together with `opath`, *i.e.* the last subpath of `path` and the first one of `opath` are joined.†
`intersect(opath, epsilon=1e-5)`	returns tuple consisting of two lists of parameter values corresponding to the intersection points of `path` and `opath`, respectively.†
`reversed()`	returns the normalized reversed `path`.†
`split(t)`	returns a tuple consisting of two `normpath`s corresponding to the `path` split at the parameter value `t`.†
`transformed(trafo)`	returns the normalized and accordingly to the linear transformation `trafo` transformed path. Here, `trafo` must be an instance of the `trafo.trafo` class.†

Some notes on the above:

- The 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.
- 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, it is a good idea to do the conversion once for performance reasons.
- Instead of using the `glue` method, you can also glue two paths together with help of the `<<` operator, for instance `p = p1 << p2`.

3.3. 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 during all of the more sophisticated path operations which are denoted by a † in the above table.

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

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

Alternatively, by passing a series of `pathel`s to the constructor, a `normpath` can be constructed like a generic `path`. The sum of a `normpath` and a `path` always yields a `normpath`.

3.4. Subclasses of path

For your convenience, some special PostScript paths are already defined, which are given in the following table.

Subclass of `path`	function
`line(x1, y1, x2, y2)`	a line from the point (`x1`, `y1`) to the point (`x2`, `y2`)
`curve(x0, y0, x1, y1, x2, y2, x3, y3)`	a Bézier curve with control points (`x0`, `y0`), ..., (`x3`, `y3`).
`rect(x, y, w, h)`	a rectangle with the lower left point (`x`, `y`), width `w`, and height `h`.
`circle(x, y, r)`	a circle with center (`x`, `y`) and radius `r`.

Note that besides the `circle` class all classes are actually subclasses of `normpath`.

4. 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.

4.1. 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 (`sy` = `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

4.2. 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 (`sy` = `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

5. Module canvas: PostScript interface

The central module 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 also various path styles (as subclasses of `base.PathStyle`), path decorations like arrows (with the class `canvas.PathDeco` and subclasses thereof), and the class `canvas.clip` which allows clipping of the output.

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.

5.1.1. Basic usage

Let us first demonstrate the basic usage of the `canvas` class. We start by constructing the main `canvas` instance, which we shall by convention always name `c`.

```
from pyx import *

c = canvas.canvas()
```

Basic drawing then proceeds via the construction of a `path`, which can subsequently be drawn on the canvas using the method `stroke()`:

```
p = path.line(0, 0, 10, 10)
c.stroke(p)
```

or more concisely:

```
c.stroke(path.line(0, 0, 10, 10))
```

You can modify the appearance of a path by additionally passing instances of the class `PathStyle`. For instance, you can draw the the above path `p` in blue:

```
c.stroke(p, color.rgb.blue)
```

Similarly, it is possible to draw a dashed version of `p`:

```
c.stroke(p, canvas.linestyle.dashed)
```

Combining of several `PathStyle`s is of course also possible:

```
c.stroke(p, color.rgb.blue, canvas.linestyle.dashed)
```

Furthermore, drawing an arrow at the begin or end of the path is done in a similar way. You just have to use the provided `barrow` and `earrow` instances:

```
c.stroke(p, canvas.barrow.normal, canvas.earrow.large)
```

Filling of a path is possible via the `fill` method of the canvas. Let us for example draw a filled rectangle

```
r = path.rect(0, 0, 10, 5)
c.fill(r)
```

Alternatively, you can use the class `filled` of the canvas module in combination with the `stroke` method:

```
c.stroke(r, canvas.filled())
```

To conclude the section on the drawing of paths, we consider a pretty sophisticated combination of the above presented `PathStyle`s:

```
c.stroke(p,
         color.rgb.blue,
         canvas.earrow.LARge(color.rgb.red,
                             canvas.stroked(canvas.linejoin.round),
                             canvas.filled(color.rgb.green)))
```

This draws the path in blue with a pretty large green arrow at the end, the outline of which is red and rounded.
A canvas may also be embedded in another one using the `insert` method. This may be useful when you want to apply a transformation on a whole set of operations. XXX: Example
After you have finished the composition of the canvas, you can write it to a file using the method `writetofile()`. It expects the required argument `filename`, the name of the output file. To write your results to the file "test.eps" just call it as follows:

```
c.writetofile("test")
```

5.1.2. Methods of the class canvas

The `canvas` class provides the following methods:

canvas method	function
`__init__(*args)`	Construct new canvas. `args` can be instances of `trafo.trafo`, `canvas.clip` and/or `canvas.PathStyle`.
`bbox()`	Returns the bounding box enclosing all elements of the canvas.
`draw(path, *styles)`	Generic drawing routine for given `path` on the canvas (*i.e.* `insert`s it together with the necessary `newpath` command, applying the given `styles`. Styles can either be instances of `base.PathStyle` or `canvas.PathDeco` (or subclasses thereof).
`fill(path, *styles)`	Fills the given `path` on the canvas, *i.e.* `insert`s it together with the necessary `newpath`, `fill` sequence, applying the given `styles`. Styles can either be instances of `base.PathStyle` or `canvas.PathDeco` (or subclasses therof).
`insert(PSOp, *args)`	Inserts an instance of `base.PSOp` into the canvas. If `args` are present, create a new `canvas`instance passing `args` as arguments and insert it. Returns `PSOp`.
`set(*styles)`	Sets the given `styles` (instances of `base.PathStyle` or subclasses) for the rest of the canvas.
`stroke(path, *styles)`	Strokes the given `path` on the canvas, *i.e.* `insert`s it togeither with the necessary `newpath`, `stroke` sequence, applying the given `styles`. Styles can either be instances of `base.PathStyle` or `canvas.PathDeco` (or subclasses thereof).
`text(x, y, text, *args)`	Inserts `text` into the canvas (shortcut for `insert(texrunner.text(x, y, text, *args))`).
`texrunner(texrunner)`	Sets the `texrunner`; default is `defaulttexrunner` from the `text` module.
`writetofile(filename, paperformat=None, rotated=0, fittosize=0, margin="1 t cm", bbox=None, bboxenlarge="1 t pt")`	Writes the canvas to `filename`. 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.

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 `writetofile` 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 to with a colour, etc.

5.3. Subclasses of base.PathStyle

The `canvas` module provides a number of subclasses of the class `base.PathStyle`, which allow to change the look of the paths drawn on the canvas. They are summarized in Appendix D.

6. Module text: TEX/LATEX interface

6.1. Basic functionality

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

- start TEX/LATEX as soon as text creation is requested
- create boxes containing the requested text on the fly
- immediately analyse the TEX/LATEX output for errors etc.
- boxes are written into the dvi output
- box extents are immediately available (they are contained in the TEX/LATEX output)
- as soon as PostScript needs to be written, stop TEX/LATEX, analyse the dvi output and generate the requested PostScript
- use Type1 fonts for the PostScript generation

Note that in order that Type1 fonts can be used by PYX, an appropriate `psfonts.map` containing entries for the used fonts has to be present in your texmf tree.

6.2. 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`[1]	access to TEX/LATEX jobname files; default: `None`; example: `("spam.aux", "eggs.log")`
`waitfortex`	wait this number of seconds for a TEX/LATEX response; default `5`
`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`
`dvicopy`	get rid of virtual fonts which PYX cannot handle (boolean); default `0`
`pyxgraphics`	enables the usage of the graphics package without further configuration (boolean); default 1
`texmessagestart`[1,2]	parsers for the TEX/LATEX start message; default: `texmessage.start`
`texmessagedocclass`[1,2]	parsers for LATEXs `\documentclass` statement; default: `texmessage.load`
`texmessagebegindoc`[1,2]	parsers for LATEXs `\begin{document}` statement; default: `(texmessage.load, texmessage.noaux)`
`texmessageend`[1,2]	parsers for TEXs `\end`/ LATEXs `\end{document}` statement; default: `texmessage.texend`
`texmessagedefaultpreamble`[1,2]	default parsers for preamble statements; default: `texmessage.load`
`texmessagedefaultrun`[1,2]	default parsers for text statements; default: `(texmessage.loadfd, texmessage.graphicsload)`

[1] The parameter might contain None, a single entry or a sequence of entries.
[2] TEX/LATEX message parsers are described in more detail below.

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

Figure 6.1.: valign example

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 one or several 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 and further parameters are attributes for this command. Those attributes might be TEX/LATEX settings as described below, TEX/LATEX message parsers as described below as well, PYX transformations, and PYX fill styles (like colors). The `text` method returns a box (see chapter 7), which can be inserted into a canvas instance by its `insert` method to get the text.
Note that for the generation of the PostScript code the TEX/LATEX instance must be terminated. 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.

6.3. TEX/LATEX settings

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 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).

Vertical alignment: `valign.top`, `valign.middle`, `valign.bottom`; when no `parbox` is defined, additionally `valign.baseline` (default); when `parbox` is defined, additionally `valign.topbaseline` (default), `valign.middlebaseline`, and `valign.bottombaseline`; see figure 6.1 for an example

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

Mathmode: `mathmode` switches the mathmode of TEX/LATEX

Font size: `size.tiny`, `size.scriptsize`, `size.footnotesize`, `size.small`, `size.normalsize` (default), `size.large`, `size.Large`, `size.LARGE`, `size.huge`, `size.Huge`

6.4. 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:

[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.

`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.

6.5. 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.

6.6. 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.

7. 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.

7.1. 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`

`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 7.1

`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

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

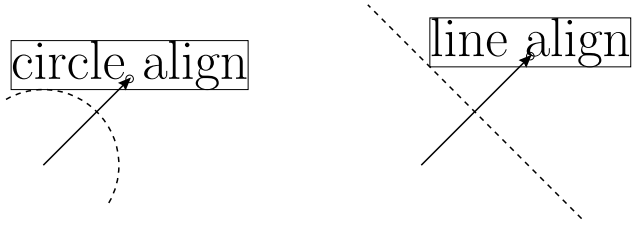


Figure 7.1.: circle and line alignment examples (equal direction and distance)

`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

7.2. 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)

7.3. 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.

8. 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]`).

8.1. Class line

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

8.2. 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.

8.3. 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.

8.4. 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`)

9. 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.writetofile("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?
`showbbox=0`	Stroke bounding box of EPS file?
`translatebox=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.

10. 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.

10.1. 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)

10.2. 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).

11. Module color

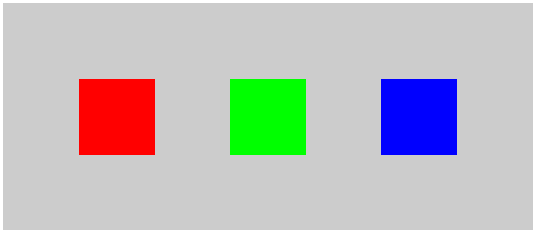
11.1. Color models

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.

11.2. 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.writetofile("color")
```

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



11.3. 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.

12. Module data

12.1. Reading a table from a file

The module datafile contains the class `datafile` which can be used to read in a table from a file. You just have to construct an instance and provide a filename as the parameter, e.g. `datafile("testdata")`. The parsing of the file, namely the columns of the table, is done by matching regular expressions. They can be modified, as they are additional named arguments of the constructor. Furthermore there is the possibility to skip some of the data points by some other keyword arguments as listed in the following table:

argument name	description
`commentpattern`	start a comment line; default: `re.compile(r"(#+|!+|%+)\s*")`
`stringpattern`	a string column; default: `re.compile(r"\"(.*?)\"(\s+|$)`
`columnpattern`	any other column; default: `re.compile(r"(.*?)(\s+|$)`
`skiphead`	skip first data lines; default: `0`
`skiptail`	skip last data lines; default: `0`
`every`	only take every `every` data line into account; default: `1`

The processing of the input file is done by reading the file line by line and first strip leading and tailing whitespaces of the line. Then a check is performed, whether the line matches the comment pattern or not. If it does match, this rest of the line is analysed like a table line when no data was read before (otherwise it is just thrown away). The result is interpreted as column titles. As the titles are sequentially overwritten by another comment line previous to the data, finally the last non-empty comment line determines the column titles.
Thus we have still to explain, how the reading of data lines works. We create a list of entries for each column out of a given line. A line resulting in an empty list (e.g. an empty line) is just ignored. As shown in the table above, there is a special string column pattern. When it matches it forces the interpretation of a column as a string. Otherwise `datafile` will try to convert the columns automatically into floats except for the title line. When the conversions fails, it just keeps the string.
The default string pattern allows for columns to contain whitespaces. It matches a string whenever it starts with a quote (") and then tries to find the end of that very string by another quote immediately followed by a whitespace or the end of the line. Hence a quote within a string is just ignored and no kind of escaping is needed. The only disadvantage is, that you cannot describe a string which contains a quote and a whitespace consecutively. However, you can always replace this string pattern to fit your special needs.

Finally the number of columns is fixed to the maximal number contained in the file and lines with less entries get filled with `None`. Also the titles list is cutted to this maximal number of columns.

12.2. Accessing columns

The method `getcolumnno` takes a parameter as the column description. If it matches exactly one entry in the titles list, the number of this element is returned. Otherwise the parameter should be an integer and it is checked, if this integer is a valid column index. Like for other python indices a column number might be negative counting the columns from the end. When an error occurres, the exception `ColumnError` is raised. Please note, that the datafile inserts a first column having the index 0, which contains the line number (starting at 1 and counting only data lines). Examples are `getcolumnno(1)` or `getcolumnno("title")`.
The method `getcolumn` takes the same argument as the method `getcolumnno` described above, but it returns a list with the values of this very column.

12.3. Mathematics on columns

By the method `addcolumn` a new column is appended. The method takes a string as the first parameter which is interpreted as an expression. When the expression contains an equal sign (`=`), everything left to the last equal sign will become the title of the new column. If no equal sign is found, the title will be set to `None`. The part right to the last equal sign is interpreted as an mathematical expression. A list of functions, predefined variables and operators can be found in appendix A. The list of available functions and predefined variables can be extended by a dictionary passed as the keyword argument `context` to the `addcolumn` method.
The expression might contain variable names. The interpretation of this names is done in the following way:

- The names can be a column title, but this is only allowed for column titles which are valid variable names (e.g. they should start with a letter or an underscore and contain only letters, digits and the underscore).
- A variable name can start with the dollar symbol (`$`) and the following integer number will directly refer to a column number.

The data referenced by variables in the expression need to be floats, otherwise the result for that data line will be `None`.

12.4. Reading data from a sectioned config file

The class `sectionfile` provides a reader for files in the ConfigFile format (see the description of the module `ConfigFile` from the pyx standard library).

12.5. Own datafile readers

The development of other datafile readers should be based on the class `data` by inheritance. When doing so, the methods `getcolumnno`, `getcolumn`, and `addcolumn` are immediately available and the cooperation with other parts of PyX is assured. All what has to be done, is a call to the inherited constructor supplying at least a sequence of data points as the `data` keyword argument. A data point itself is a sequence of floats and/or strings. Additionally a sequence of column titles (strings) might be given in the `titles` argument.

13. Module graph: graph plotting

13.1. Introductory notes

The graph module is considered to be in constant, gradual development. For the moment we concentrate ourself on standard 2d xy-graphs taking all kind of possible specialties into account like any number of axes. Architectural decisions play the most substantial role at the moment and have hopefully already been done that way, that their flexibility will suffice for future usage in quite different graph applications, *e.g.* circular 2d graphs or even 3d graphs. We will describe those parts of the graph module here, which are in a totally usable state already and are hopefully not to be changed later on. However, future developments certainly will cause some incompatibilities. At least be warned: Nobody knows the hole list of things that will break. At the moment, keeping backwards compatibility in the graph module is not at all an issue. Although we do not yet claim any backwards compatibility for the future at all, the graph module is certainly one of the biggest construction sites within PyX.
The task of drawing graphs is splitted in quite some subtasks, which are implemented by classes of its own. We tried to make those components as independend as it is usefull and possible in order to make them reuseable for different graph types. They are also replaceable by the user to get more specialized graph drawing tasks done without needing to implement a hole graph system. A major abstraction layer are the so-called graph coordinates. Their range is generally fixed to $[0;1]$. Only the graph does know about the conversion between these coordinates and the position at the canvas (the graph itself is its canvas, that can be inserted into another canvas). By that, all other components can be reused for different graph geometries. The interfaces between the components are documented in docstrings of interface classes in the source. The interface technique is used for documentation purposes only. For the user, the most important informations (as described in the following) are the parameters of the constructors of the various implementations of those interface. They are documented in docstrings of the constructors of the classes. While effort of clearing and documenting is still in progress, some parts are already nicely documented as you may see by using `pydoc`.

13.2. Axes

A common feature of a graph are axes. An axis is responsible for the conversion of values to graph coordinates. There are predefined axis types, namely:

axis type	description
`linaxis`	linear axis
`logaxis`	logarithmic axis

Further axes types are available to support axes splitting and bar graphs (other axes types might be added in the future as well), but they behave quite different from the generic case and are thus described separately below.

13.2.1. Axes properties

Global properties of an axis are set as named parameters in the axis constructor. Both, the `linaxis` and the `logaxis`, have the same set of named parameters listed in the following table:

argument name	description
`title`	axis title
`min`	fixes axis minimum; if not set, it is automatically determined, but this might fail, for example for the x-range of functions, when it is not specified there
`max`	as above, but for the maximum
`reverse`	boolean; exchange minimum and maximum (might be used without setting minimum and maximum); if min¿max and reverse is set, they cancel each other
`divisor`	numerical divisor for the axis partitioning; default: `1`
`suffix`	a suffix to indicate the divisor within an automatic axis labeling
`datavmin`	minimal graph coordinate when adjusting the axis minima to the graph data; default: `0.05` (or `0`, when min is present)
`datavmax`	as above, but for the maximum; default: `0.95` (or `1`, when max is present)
`tickvmin`	minimal graph coordinate for placing ticks to the axis; default: `0`
`tickvmax`	as above, but for the maximum; default: `1`
`density`	density parameter for the axis partition rating
`maxworse`	number of trials with worse tick rating before giving up; default: `2`
`painter`	axis painter (described below)
`texter`	texter for the axis labels (described below)
`part`	axis partition (described below)
`rater`	partition rater (described below)

13.2.2. Partitioning of axes

The definition of ticks and labels appropriate to an axis range is called partitioning. The axis partioning within PyX uses rational arithmetics, which avoids any kind of rounding problems to the cost of performance. The class `frac` supplies a rational number. It can be initialized by another `frac` instance, a tuple of integers (called enumerator and denominator), a float (which gets converted into a frac with a finite resolution `floatprecision`

of 10 digits per default) or a string with infinite precision (like "1.2345e-100" or even "1/3"). However, a partitioning is composed out of a sorted list of ticks, where the class `tick` is derived from `frac` and has additional properties called `ticklevel`, `labellevel`, `label` and `labelattrs`. When the ticklevel or the labellevel is `None`, it just means not present, `0` means tick or label, respectively, `1` means subtick or sublevel and so on. When `labellevel` is not `None`, a `label` might be explicitly given, which will get used as the text of that label. Otherwise the axis painter has to create an appropriate text for the label. The `labelattrs` might specify some attributes for the label to be used by the `text` method of an `texrunner` instance.
You can pass a list of `tick` instance to the `part` argument of an axis. By that you can place ticks whereever you want. (In former versions there was a manual partitioner and the possibility of mixing partitions for that. This is still available in this version of PyX, but it will be removed in the future.) Additionally you can use a partitioner to create ticks appropriate to the axis range. This can be done by manually specifying distances between ticks, subticks and so on. Alternatively there are automatic axis partitioners available, which provide different partitions and the rating of those different partitions by the `rater` become crutial. Note, that you can combine manually set ticks and a partitioner in the `part` argument of an axis.

Partitioning of linear axes

The class `linpart` creates a linear partition as described by named parameters of the constructor:

argument name	default	description
`tickdist`	`None`	distance between ticks, subticks, etc. (see comment below); when the parameter is `None`, ticks will get placed at labels
`labeldist`	`None`	distance between labels, sublabels, etc. (see comment below); when the parameter is `None`, labels will get placed at ticks
`labels`	`None`	set the text for the labels manually
`extendtick`	`0`	allow for a range extention to include the next tick of the given level
`extendlabel`	`None`	as above, but for labels
`epsilon`	`1e-10`	allow for exceeding the range by that relative value
`mix`	`()`	an obsolete feature to mix-in additional ticks (to be removed in future versions)

The `ticks` and `labels` can either be a list or just a single entry. When a list is provided, the first entry stands for the tick or label, respectively, the second for the subtick or sublabel, and so on. The entries are passed to the constructor of a frac instance, e.g. there can be tuples (enumerator, denominator), strings, floats, etc.

Partitioning of logarithmic axes

The class `logpart` create a logarithmic partition. The parameters of the constructor of the class `logpart` are quite simular to the parameters of `linpart` discussed above. Instead of the parameters `tickdist` and `labeldist` the parameters `tickpos` and `labelpos` are present. All other parameters of `logpart` do not differ in comparision to `linpart`. The `tickdist` and `labeldist` parameters take either a single `preexp` instance or a list of `preexp` instances, where the first stands for the ticks (labels), the second for subticks (sublabels) and so on. A `preexp` instance contains a list of `pres`, which are `frac` instances defining some positions, say p_i. Furthermore there is a `frac` instance called `exp`, say e. Valid tick and label positions are then given by $s^n p_i$, where n is an integer.

name	values it descibes
`pre1exp5`	1 and multiple of 10^5
`pre1exp4`	1 and multiple of 10^4
`pre1exp3`	1 and multiple of 10^3
`pre1exp2`	1 and multiple of 10^2
`pre1exp`	1 and multiple of 10
`pre125exp`	1, 2, 5 and multiple of 10
`pre1to9exp`	1, 2, ..., 9 and multiple of 10

Automatic partitioning of linear axes

When no explicit axis partitioner is given in the constructor argument `part` of an linear axis, it is initialized with an automatic partitioning scheme for linear axes. This scheme is provided by the class `autolinpart`, where the constructor takes the following arguments:

argument name	default	description
`variants`	`defaultvariants`	list of possible values for the ticks parameter of `linpart` (labels are placed at the position of ticks)
`extendtick`	`0`	allow for a range extention to include the next tick of the given level
`epsilon`	`1e-10`	allow for exceeding the range by that relative value
`mix`	`()`	as in linpart

The default value for the argument `variants`, namely `defaultvariants`, is defined as a class variable of `autolinpart` and has the value `((frac(1, 1), frac(1, 2)), (frac(2, 1), frac(1, 1)), (frac(5, 2), frac(5, 4)), (frac(5, 1), frac(5, 2)))`. This implies, that the automatic axis partitioning scheme allows for partitions using (ticks, subticks) with at distances (1, 1/2), (2, 1), (5/2, 5/4), (5, 5/2). This list must be sorted by the number of ticks the entries will lead to. The given fractions are automatically multiplied or divided by 10 in order to fit better to the axis range. Therefore those additional partitioning possibilities (infinte possibilities) must not be given explicitly.

Automatic partitioning of logarithmic axes

When no explicit axis partitioning is given in the constructor argument `part` of an logarithmic axis, it is initialized with an automatic partitioning schemes for logarithmic axes. This scheme is provided by the class `autologpart`, where the constructor takes the following arguments:

argument name	default	description
`variants`	`defaultvariants`	list of pairs with possible values for the ticks and labels parameters of `logpart`
`extendtick`	`0`	allow for a range extention to include the next tick of the given level
`extendlabel`	`None`	as above, but for labels
`epsilon`	`1e-10`	allow for exceeding the range by that relative value
`mix`	`()`	as in linpart

The default value for the argument `variants`, namely `defaultvariants`, is defined as a class variable of `autologpart` and has the value:

```
(((pre1exp, pre1to9exp),        # ticks & subticks,
         (pre1exp, pre125exp)), # labels & sublevels
 ((pre1exp, pre1to9exp), None), # ticks & subticks, labels=ticks
 ((pre1exp2, pre1exp), None),   # ticks & subticks, labels=ticks
 ((pre1exp3, pre1exp), None),   # ticks & subticks, labels=ticks
 ((pre1exp4, pre1exp), None),   # ticks & subticks, labels=ticks
 ((pre1exp5, pre1exp), None))   # ticks & subticks, labels=ticks
```

As for the `autolinaxis`, this list must be sorted by the number of ticks the entries will lead to.

Rating of axes partitionings

When an axis partitioning scheme returns several partitioning possibilities, the partitions are rated by an instance of a rater class provided as the parameter `rater` at the axis constructor. It is used to calculate a positive rating number for a given axis partitioning. In the end, the lowest rated axis partitioning gets used.
The rating consists of two steps. The first takes into account only the number of ticks, subticks, labels and so on in comparison to an optimal number. Additionally, the transgression 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 the layout of a partition gets acknowledged by rating the distance of the labels to each other. Thereby partitions with overlapping labels get quashed out.
The class `axisrater` implements a rating with quite some parameters specifically adjusted to linear and logarithmic axes. A detailed description of the hole system goes beyond the scope of that manual. Take your freedom and have a look at the PyX source code if you wish to adopt the rating to personal preferences.

The overall optimal partition properties, namely the density of ticks and labels, can be easily adjusted by the single parameter `density` of the axis constructor. The rating is adjusted to the default densitiy value of `1`, but modifications of this parameter in the range of 0.5 (for less ticks) to 2 or even 3 (for more ticks) might be usefull.

13.2.3. Creating label text

When a partition is created, the typical situation is that some of the ticks have a `labellevel` not equal to `None` but there is no `label` (a string) defined to be printed at this tick. The task of a `texter` is to create those label strings for a given list of ticks. There are different `texter` classes creating different label strings.

Decimal numbers

The class `decimaltexter` creates decimal labels. The format of the labels can be configured by numerous arguments of the constructor listed in the following table:

argument name	default	description
`prefix`	`""`	string to be inserted in front of the number
`infix`	`""`	string to be inserted between the plus or minus sign and the number
`suffix`	`""`	string to be inserted after the number
`equalprecision`	`0`	forces a common number of digits after the comma
`decimalsep`	`"."`	decimal separator
`thousandsep`	`""`	thousand separator
`thousandthpartsep`	`""`	thousandth part separator
`plus`	`""`	plus sign
`minus`	`"-"`	minus sign
`period`	`r"\overline{%s}"`	format string to indicate a period
`labelattrs`	`text.mathmode`	a single attribute or a list of attributes to be added to the labelattrs

Decimal numbers with an exponential

The class `exponentialtexter` creates decimal labels with an exponent. The format of the labels can be configured by numerous arguments of the constructor listed in the following table:

argument name	default	description
`plus`	`""`	plus sign for the exponent
`minus`	`"-"`	minus sign for the exponent
`mantissaexp`	`r"{{%s}\cdot10^{%s}}"`	format string for manissa and exponent
`nomantissaexp`	`r"{10\{%s}}"`	format string when skipping a manissa equals 1
`minusnomantissaexp`	`r"{-1\{%s}}"`	format string when skipping a manstissa equals -1
`mantissamin`	`frac((1, 1))`	minimal value for the mantissa
`mantissamax`	`frac((10, 1))`	maximal value for the mantissa
`skipmantissa1`	`0`	skip mantissa equals 1
`skipallmantissa1`	`1`	skip mantissa when its always 1
`mantissatexter`	`decimaltexter()`	texter for the mantissa

Decimal numbers without or with an exponential

The class `defaulttexter` creates decimal labels without or with an exponent. As the name says, its used as the default texter. The texter splits the tick list into two lists, one to be passed to a decimal texter and another to be passed to an exponential texter. This splitting is based on the two properties `smallestdecimal` and `biggestdecimal`. See the following table for all available arguments:

argument name	default	description
`smallestdecimal`	`frac((1, 1000))`	the smallest number (ignoring the sign) where the decimal texter should be used
`biggestdecimal`	`frac((9999, 1))`	as above, but for the biggest number
`equaldecision`	`1`	either use the `decimaltexter` or the `exponentialtexter`
`decimaltexter`	`decimaltexter()`	texter withoud an exponential
`exponentialtexter`	`exponentialtexter()`	exponential with an exponential

Rational numbers

The class `rationaltexter` creates rational labels. The format of the labels can be configured by numerous arguments of the constructor listed in the following table:

argument name	default	description
`prefix`	`""`	string to be inserted in front of the rational
`infix`	`""`	string to be inserted between the plus or minus sign and the rational
`suffix`	`""`	string to be inserted after the rational
`enumprefix`	`""`	as `prefix` but for the enumerator
`enuminfix`	`""`	as `infix` but for the enumerator
`enumsuffix`	`""`	as `suffix` but for the enumerator
`denomprefix`	`""`	as `prefix` but for the denominator
`denominfix`	`""`	as `infix` but for the denominator
`denomsuffix`	`""`	as `suffix` but for the denominator
`plus`	`""`	plus sign
`minus`	`"-"`	minus sign
`minuspos`	`0`	position of the minus: `0` – in front of the fraction, `1` – in front of the enumerator, `-1` – in front of the denominator
`over`	`r"{{%s}\over{%s}}"`	format string for the fraction
`equaldenom`	`0`	usually, the enumerator and the denominator are canceled; if set, take the least common multiple of all denominators
`skip1`	`1`	skip printing the fraction for `1` when there is a `prefix`, `infix`, or `suffix`
`skipenum0`	`1`	print `0` instead of a fraction when the enumerator is 0
`skipenum1`	`1`	as `skip1` but for the enumerator
`skipdenom1`	`1`	skip the denominator when it is `1` and there is no `denomprefix`, `denominfix`, or `denomsuffix`
`labelattrs`	`text.mathmode`	a single attribute or a list of attributes to be added to the labelattrs

13.2.4. Painting of axes

A major task for an axis is its painting. It is done by instances of `axispainter`, provided to the constructor of an axis as its `painter` argument. The constructor of the axis painter receives a numerous list of named parameters to modify the axis look. A list of parameters is provided in the following table:

argument name	description
`innerticklengths`[1,4]	tick length of inner ticks (visual length); default: `axispainter.defaultticklengths`
`outerticklengths`[1,4]	as before, but for outer ticks; default: `None`
`tickattrs`[2,4]	stroke attributes for ticks; default: `()`
`gridattrs`[2,4]	stroke attributes for grid lines; default: `None`
`zerolineattrs`[3,4]	stroke attributes for a grid line at axis value 0; default: `()`
`baselineattrs`[3,4]	stroke attributes for the axis baseline; default: `canvas.linecap.square`
`labeldist`	label distance from axis (visual length); default: `"0.3 cm"`
`labelattrs`[2,4]	text attributes for labels; default: `(text.halign.center, text.vshift.mathaxis)`
`labeldirection`[4]	relative label direction (see below); default: `None`
`labelhequalize`	set width of labels to its maximum (boolean); default: `0`
`labelvequalize`	set height and depth of labels to their maxima (boolean); default: `1`
`titledist`	title distance from labels (visual length); default: `"0.3 cm"`
`titleattrs`[3,4]	text attributes for title; default: `(text.halign.center, text.vshift.mathaxis)`
`titledirection`[4]	relative title direction (see below); default: `paralleltext`
`titlepos`	title position in graph coordinates; default: `0.5`

[1] The parameter should be a list, where the entries are attributes for the different levels. When the level is larger then the list length, `None` is assumed. When the parameter is not a list, it is applied to all levels.
[2] The parameter should be a list of lists, where the entries are attributes for the different levels. When the level is larger then the list length, `None` is assumed. When the parameter is not a list of lists, it is applied to all levels.
[3] The parameter should be a list. When the parameter is not a list, the parameter is interpreted as a list with a single entry.
[4] The feature can be turned off by the value `None`. Within lists or lists of lists, the value `None` might be used to turn off the feature for some levels selectively.

Relative directions for labels (`labeldirection`) and titles (`titledirection`) are instances of `rotatetext`. By that the text direction is calculated relatively to the tick direction of the axis and is added as an attribute of the text. The relative direction provided by instances of the class `rotatetext` prevent upside down text by flipping it by 180 degrees. For convenience, the two self-explanatory values `rotatetext.parallel` and `rotatetext.orthogonal` are available, which are just instances of rotatetext initializes by `-90` degree and `0`, respectively.

13.2.5. Linked axes

Linked axes can be used whenever an axis should be repeated within a single graph or even between different graphs although the intrinsic meaning is to have only one axis plotted several times. Almost all properties of a linked axis are supplied by the axis it is linked to (you may call it the base axis), but some properties and methods might be different. For the typical case (implemented by `linkaxis`) only the painter of the axis is exchanged together with some simplified behaviour when finishing the axis (there is no need to recalculate the axis partition etc.). The constructor of `linkaxis` takes the axis to be linked to as the first parameter and in the named parameter `painter` a new painter for the axis. By default, `linkaxispainter` is used, which differs from the standard `axispainter` by some default values for the arguments only. Namely, the arguments `zerolineattrs`, `labelattrs`, and `titleattrs` are set to `None` turing off these features.

13.2.6. Special purpose axes

Splitable axes

Axes with breaks are created by instances of the class `splitaxis`. Its constructor takes the following parameters:

argument name	description
(axis list)	a list of axes to be used as subaxes (this is the first parameter of the constructor; it has no name)
`splitlist`	a single number or a list split points of the possitions of the axis breaks in graph coordinates; the value `None` forces `relsizesplitdist` to be used; default: `0.5`
`splitdist`	gap of the axis break; default: `0.1`
`relsizesplitdist`	used when `splitlist` entries are `None`; gap of the axis break in values of the surrounding axes (on logarithmic axes, a decade corresponds to 1); the split position is adjusted to give both surrounding axes the same scale (thus, their range must be completely fixed); default: `1`
`title`	axis title
`painter`	axis painter; default: `splitaxispainter()` (described below)

A split axis is build up from a list of "subaxes". Those subaxes have to provide some range information needed to identify the subaxis to be used out of a plain number (thus all axes minima and maxima has to be set except for the two subaxes at the egde, where for the first only the maximum is needed, while for the last only the minimum is needed). The only point left is the description of the specialized `splitaxispainter`, where the constructor takes the following parameters:

argument name	description
`breaklinesdist`	(visual) distance between the break lines; default: `0.05`
`breaklineslength`	(visual) length of break lines; default: `0.5`
`breaklinesangle`	angle of the breakline with respect to the axis; default: `-60`
`breaklinesattrs`	stroke attributes for the break lines (`None` to turn off the break lines, otherwise a single value or a tuple); default: `()`

Additionally, the painter takes parameters for the axis title formatting like the standard axis painter class `axispainter`. (There is a common base class `titleaxispainter` for that.) The parameters are `titledist`, `titleattrs`, `titledirection`, and `titlepos`.

Bar axes

Axes appropriate for bar graphs are created by instances of the class `baraxis`. Its constructor takes the following parameters:

argument name	description
`subaxis`	baraxis can be recursive by having another axis as its subaxis; default: `None`
`multisubaxis`	contains another baraxis instance to be used to construct a new subaxes for each item (by that a nested bar axis with a different number of subbars at each bar can be created) ; default: `None`
`title`	axis title
`dist`	distance between bars (relative to the bar width); default: `0.5`
`firstdist`	distance of the first bar to the border; default: `0.5*dist`
`lastdist`	as before but for the last bar
`names`	tuple of name identifiers for bars; when set, no other identifiers are allowed; default: `None`
`texts`	dictionary translating names into label texts (otherwise just the names are used); default: {}
`painter`	axis painter; default: `baraxispainter` (described below)

In contrast to other axes, a bar axis uses name identifiers to calculate a position at the axis. Usually, a style appropriate to a bar axis (this is right now just the bar style) set those names out of the data it recieves. However, the names can be forced and fixed.
Bar axes can be recursive. Thus for a given value, an appropriate subaxis is choosen (usually another bar axis). Usually only a single subaxis is needed, because it doesn't need to be painted and for each value the same recursive subaxis transformation has to be applied. This is achieved by using the parameter `subaxis`. Alternatively you may use the `multisubaxis`. Here only a bar axis can be used. Then the subaxes (note axes instead of axis) are painted as well (however their painter can be set to not paint anything). For that, duplications of the subaxis are created for each name. By that, each subaxis can have different names, in particular different number of names.
The only point left is the description of the specialized `baraxispainter`. It works quite similar to the `axispainter`. Thus the constructors have quite some parameters in common, namely `titledist`, `titleattrs`, `titledirection`, `titlepos`, and

baselineattrs. Furthermore the parameters innerticklength and outerticklength work like their counterparts in the axispainter, but only plain values are allowed there (no lists). However, they are both None by default and no ticks get plotted. Then there is a hole bunch of name attribute identifiers, namely namedist, nameattrs, namedirection, namehequalize, namevequalize which are identical to their counterparts called label... instead of name.... Last but not least, there is a parameter namepos which is analogous to titlepos and set to 0.5 by default.

13.3. Data

13.3.1. List of points

Instances of the class data link together a datafile (or another instance of a class from the module data) and a style (see below; default is symbol). The link object is needed in order to be able to plot several data from a singe file without reading the file several times. However, for easy usage, it is possible to provide just a filename instead of a datafile instance as the first argument to the constructor of the class data thus hiding the underlying datafile instance completely from view. This is the preferable solution as long as the datafile gets used only once.
The additional parameters of the constructor of the class data are named parameters. The values of those parameters describe data columns which are linked to the names of the parameters within the style. The data columns can be identified directly via their number or title, or by means of mathematical expression (as in the addcolumn method of the class data in the module data; see chapter 12; indeed a addcolumn call takes place to evaluate mathematical expressions once and for all).
The constructors keyword argument title however does not refer to a parameter of a style, but instead sets the title to be used in an axis key.

13.3.2. Functions

The class function provides data generation out of a functional expression. The default style for function plotting is line. The constructor of function takes an expression as the first parameter. The expression must be a string with exactly one equal sign (=). At the left side the result axis identifier must be placed and at the right side the expression must depend on exactly one variable axis identifier. Hence, a valid expression looks like "y=sin(x)". You can access own variables and functions by providing them as a dictionary to the constructors context keyword argument.
Additional named parameters of the constructor are:

argument name	default	description
`min`	`None`	minimal value for the variable parameter; when `None`, the axis data range will be used
`max`	`None`	as above, but for the maximum
`points`	`100`	number of points to be calculated
`parser`	`mathtree.parser()`	parser for the mathematical expression
`context`	`None`	dictionary of extern variables and functions
`title`	equal to the expression	title to be used in the graph key

The expression evaluation takes place at a linear raster on the variable axis. More advanced methods (detection of rapidely changing functions, handling of divergencies) are likely to be added in future releases.

13.3.3. Parametric functions

The class `paramfunction` provides data generation out of a parametric representation of a function. The default style for parametric function plotting is `line`. The parameter list of the constructor of `paramfunction` starts with three parameters describing the function parameter. The first parameter is a string, namely the variable name. It is followed by a minimal and maximal value to be used for that parameter. The next parameter contains an expression assigning functions to the axis identifiers in a quite pythonic tuple notation. As an example, such an expression could look like `"x, y = sin(k), cos(3*k)"`. Additionally, the named parameters `points`, `parser`, `context`, and `title` behave like their equally named counterparts in `function`.

13.4. Styles

Styles are used to draw data at a graph. A style determines what is painted and how it is painted. Due to this powerfull approach there are already some different styles available and the possibility to introduce other styles opens even more prospects.

13.4.1. Symbols

The class `symbol` can be used to plot symbols, errorbars and lines configurable by parameters of the constructor. Providing `None` to attributes hides the according component.

argument name	default	description
`symbol`	`changesymbol.cross()`	symbol to be used (see below)
`size`	`"0.2 cm"`	size of the symbol (visual length)
`symbolattrs`	`canvas.stroked()`	draw attributes for the symbol
`errorscale`	`0.5`	size of the errorbar caps (relative to the symbol size)
`errorbarattrs`	`()`	stroke attributes for the errorbars
`lineattrs`	`None`	stroke attributes for the line

The parameter `symbol` has to be a routine, which returns a path to be drawn (e.g. stroked or filled). There are several such routines already available in the class `symbol`, namely `cross`, `plus`, `square`, `triangle`, `circle`, and `diamond`. Furthermore, changeable attributes might be used here (like the default value `changesymbol.cross`), see section 13.4.7 for details.
The attributes are available as class variables after plotting the style for outside usage. Additionally, the variable `path` contains the path of the line (even when it wasn't plotted), which might be used to get crossing points, fill areas, etc.
Valid data names to be used when providing data to symbols are listed in the following table. The character `X` stands for axis names like `x`, `x2`, `y`, etc.

data name	description
`X`	position of the symbol
`Xmin`	minimum for the errorbar
`Xmax`	maximum for the errorbar
`dX`	relative size of the errorbar: `Xmin, Xmax = X-dX, X+Xd`
`dXmin`	relative minimum `Xmin = X-dXmin`
`dXmax`	relative maximum `Xmax = X+dXmax`

13.4.2. Lines

The class `line` is inherited from the class `symbol` and restricted itself to line drawing. The constructor takes only the `lineattrs` keyword argment, which is by default set to `(changelinestyle(), canvas.linejoin.round)`. The other features of the symbol style are turned off.

13.4.3. Rectangles

The class `rect` draws filled rectangles into a graph. The size and the position of the rectangles to be plotted can be provided by the same data names like for the errorbars of the class `symbol`. Indeed, the class `symbol` reuses most of the symbol code by inheritance, while modifying the errorbar look into a colored filled rectangle and turing off the symbol itself.
The color to be used for the filling of the rectangles is taken from a palette provided to the constructor by the named parameter `palette` (default is `color.palette.Gray`). The data name `color` is used to select the color out of this palette.

13.4.4. Texts

Another style to be used within graphs is the class `text`, which adds the output of text to the class `symbol`. The text position relative to the symbol is defined by the two named parameters `textdx` and `textdy` having a default of `"0 cm"` and `"0.3 cm"`, respectively, which are by default interpreted as visual length. A further named parameter `textattrs` may contain a list of text attributes (or just a single attribute). The default for this parameter is `text.halign.center`. Furthermore the constructor of this class allows all other attributes of the class `symbol`.

13.4.5. Arrows

The class `arrow` can be used to plot small arrows into a graph where the size and direction of the arrows has to be given within the data. The constructor of the class takes the following parameters:

argument name	default	description
`linelength`	`"0.2 cm"`	length of a the arrow line (visual length)
`arrowattrs`	`()`	stroke attributes
`arrowsize`	`"0.1 cm"`	size of the arrow (visual length)
`arrowdict`	{}	attributes to be used in the `earrow` constructor
`epsilon`	1e-10	smallest allowed arrow size factor for a arrow to become plotted (avoid numerical instabilities)

The arrow allows for data names like the symbol and introduces additionally the data names `size` for the arrow size (as an multiplicator for the sizes provided to the constructor) and `angle` for the arrow direction (in degree).

13.4.6. Bars

The class `bar` must be used in combination with an `baraxis` in order to create bar plots. The constructor takes the following parameters:

argument name	description
`fromzero`	bars start at zero (boolean); default: `1`
`stacked`	stack bars (boolean/integer); for values bigger than 1 it is the number of bars to be stacked; default: `0`
`skipmissing`	skip entries in the bar axis, when datapoints are missing; default: `1`
`xbar`	bars parallel to the graphs x-direction (boolean); default: `0`
`barattrs`	fill attributes; default: `(canvas.stroked(color.gray.black), changecolor.Rainbow())`

Additionally, the bar style takes two data names appropriate to the graph (like `x`, `x2`, and `y`).

13.4.7. Iterateable style attributes

The attributes provided to the constructors of styles can usually handle so called iterateable attributes, which are changing itself when plotting several data sets. Iterateable attributes can be easily written, but there are already some iterateable attributes available for the most common cases. For example a color change is done by instances of the class `colorchange`, where the constructor takes a palette. Applying this attribute to a style and using this style at a list of data, the color will get changed lineary along the palette from one end to the other. The class `colorchange` includes inherited classes as class variables, which are called like the color palettes shown in appendix C. For them the default palette is set to the appropriate color palette.
Another attribute changer is called `changesequence`. The constructor takes a list of attributes and the attribute changer cycles through this list whenever a new attribute is requested. This attribute changer is used to implement the following attribute changers:

attribute changer	description
`changelinestyle`	iterates linestyles solid, dashed, dotted, dashdotted
`changestrokedfilled`	iterates `(canvas.stroked(), canvas.filled())`
`changefilledstroked`	iterates `(canvas.filled(), canvas.stroked())`

The class `changesymbol` can be used to cycle throu symbols and it provides already various specialized classes as class variables. To loop over all available symbols (cross, plus, square, triangle, circle, and diamond) the equal named class variables can be used. They start at that symbol they are named of. Thus `changesymbol.cross()` cycles throu the list starting at the cross symbol. Furthermore there are four class variables called `squaretwice`, `triangletwice`, `circletwice`, and `diamondtwice`. They cycle throu the four fillable symbols, but returning the symbols twice before they go on to the next one. They are intented to be used in combination with `changestrokedfilled` and `changefilledstroked`.

13.5. Keys

Graph keys can be created by instances of the class `key`. Its constructor takes the following keyword arguments:

argument name	description
`dist`	(vertical) distance between the key entries (visual length); default: `"0.2 cm"`
`pos`	`"tr"` (top right; default), `"br"` (bottom right), `"tl"` (top left), `"bl"` (bottom left)
`hdist`	horizontal distance of the key (visual length); default: `"0.6 cm"`
`vdist`	vertical distance of the key (visual length); default: `"0.4 cm"`
`hinside`	align horizonally inside to the graph (boolean); default: `1`
`vinside`	align vertically inside to the graph (boolean); default: `1`
`symbolwidth`	width reserved for the symbol (visual length); default: `"0.5 cm"`
`symbolheight`	height reserved for the symbol (visual length); default: `"0.25 cm"`
`symbolspace`	distance between symbol and text (visual length); default: `"0.2 cm"`
`textattrs`	text attributes (a list or a single entry); default: `text.vshift.mathaxis`

The data description to be printed in the graph key is given by the title of the data drawn.

13.6. X-Y-Graph

The class `graphxy` draws standard x-y-graphs. It is a subcanvas and can thus be just inserted into a canvas. The x-axes are named `x`, `x2`, `x3`, ... and equally the y-axes. The number of axes is not limited. All odd numbered axes are plotted at the bottom (for x axes) and at the left (for y axes) and all even numbered axes are plotted opposite to them. The lower numbers are closer to the graph.
The constructor of `graphxy` takes axes as named parameters where the parameter name is an axis name as just described. Those parameters refer to an axis instance as they where described in section 13.2. When no `x` or `y` is provided, they are automatically set to instances of `linaxis`. When no `x2` or `y2` axes are given they are initialized as standard linkaxis to the axis `x` and `y`. However, you can turn off the automatism by setting those axes explicitly to `None`.
However, the constructor takes some more attributes listed in the following table:

argument name	default	description
`xpos`	`"0"`	x position of the graph (user length)
`ypos`	`"0"`	y position of the graph (user length)
`width`	`None`	width of the graph area (axes are outside of that range)
`height`	`None`	as abovem, but for the height
`ratio`	`goldenrule`	width/height ratio when only a width or height is provided
`backgroundattrs`	`None`	background attributes for the graph area
`axisdist`	`"0.8 cm"`	distance between axis (visual length)
`key`	`None`	`key` instance for an automatic graph key

After a graph is constructed, data can be plotted via the `plot` method. The first argument should be an instance of the data providing classes described in section 13.3. This first parameter can also be a list of those instances when you want to iterate the style you explicitly provide as a second parameter to the plot method. The plot method returns the plotinfo instance (or a list of plotinfo instances when a data list was provided). The plotinfo class has attributes `data` and `style`, which provide access to the plotted data and the style, respectively. Just as an example, from the style you can access the path of the drawed line, fill areas with it etc.
After the plot method was called once or several times, you may explicitly call the method `finish`. Most of the graphs functionallity becomes available just after (partially) finishing the graph. A partial finish can even modify the order in which a graph performs its drawing process. By default the five methods `dolayout`, `dobackground`, `doaxis`, `dodata`, and `dokey` are called in that order. The method `dolayout` must always be called first, but this is internally ensured once you call any of the routines yourself. After `dolayout` gets called, the method `plot` can not be used anymore.
To get a position within a graph as a tuple out of some axes values, the method `pos` can be used. It takes two values for a position at the x and y axis. By default, the axes named `x` or `y` are used, but this is changed when the keyword arguments `xaxis` and `yaxis` are set to other axes. The graph axes are available by their name using the dictionary `axes`. Each axis has a method `gridpath` which is set by the graph. It returns a path of a grid line for a given position at the axis.
To manually add a graph key, use the `addkey` method, which takes a `key` instance first followed by several `plotinfo` instances.

14. Module tex: TEX/LATEX interface (obsolete)

Please note: THIS MODULE IS OBSOLETE. Consider the `text` module instead.

14.1. Methods

Text in PYX is created by TEX or LATEX. From the technical point of view, the text is inserted as an Encapsulated PostScript file (`eps`-file). This `eps`-file is generated by the module `tex` which runs TEX or LATEX followed by `dvips` to create the requested text. TEX is used by instances of the class `tex` while LATEX is used by `latex`. Up to the constructor and the advanced possibilities in LATEX commands both classes `tex` and `latex` are identical. They provide 5 methods to the user listed in the following table:

method	task	allowed attributes in `*attr`
`text(x, y, cmd, *attr)`	print `cmd`	`style, fontsize, halign, valign, direction, color, msghandler(s)`
`define(cmd, *attr)`	execute `cmd`	`msghandler(s)`
`textwd(cmd, *attr)`	width of `cmd`	`style, fontsize, missextents, msghandler(s)`
`textht(cmd, *attr)`	height of `cmd`	`style, fontsize, valign, missextents, msghandler(s)`
`textdp(cmd, *attr)`	depth of `cmd`	`style, fontsize, valign, missextents, msghandler(s)`

There are some common rules:

- `cmd` stands for a TEX or LATEX expression. To prevent a backslash plague, python's raw string feature can nicely be used. `x`, `y` specify a position.
- `define` can only be called before any of the other methods. In LATEX definitions are inserted directly in front of the `\begin{document}` statement. However, this is not a limitation, because by `\AtBeginDocument{}` definitions can be postponed.
- The extent routines `textwd`, `textht`, and `textdp` return true PYX length (see section 2). Usually, the evaluation takes place when performing a write and the results are stored in a file with the suffix `.size`. Therefore you have to run your file twice at first to get the correct value. This default behaviour can be changed by the `missextents` attribute.

- All commands are passed to TEX or LATEX in the calling order of the methods with one exception: if the same command is used several times (for printing as well as for calculating extents), all requests are executed at the position of the first occurrence of the command.
- All text is inserted into the `canvas` at the position, where the `tex-` or `latex-` instance itself is inserted into the `canvas`. In fact, the `eps`-file created by TEX or LATEX and `dvips` is just inserted.
- The tailing `*style` parameter stands for a list of attribute parameters listed in the last column of the table. Attribute parameters are instances of classes discussed in detail in the following section.
- There can be several `msghandler` attributes which will be applied sequentially. All other parameters can occure only once.

14.2. Attributes

`style:` `style.text` (default – does nothing to the command),
`style.math` (switches to math mode in `\displaystyle`)

`fontsize:` specifies the LATEX font sizes by `fontsize.xxx` where `xxx` is one of `tiny`, `scriptsize`, `footnotesize`, `small`, `normalsize` (default), `large`, `Large`, `LARGE`, `huge`, or `Huge`.

`halign:` `halign.left` (default), `halign.center`, `halign.right`

`valign:` `valign.top(length)` or `valign.bottom(length)` — creates a vertical box with width `length`. The vertical alignment is the baseline of the first line for `top` and the last line for `bottom`. The box width is stored in the TEX dimension `\linewidth`.

`direction:` `direction.xxx` where `xxx` stands for `horizontal` (default), `vertical`, `upsidedown`, or `rvertical`. Additionally, any angle `angle` (in degree) is allowed in `direction(angle)`.

`color:` stands for any PYX color (see section 11), default is `color.gray.black`

`missextents:` provides a routine, which is called when a requested extent is not yet available. In the following table a list of choises for this parameter is described:

missextents	description
`missextents.returnzero`	returns zero (default)
`missextents.returnzeroquiet`	as above, but does not return a warning via `atexit`
`missextents.raiseerror`	raise `TexMissExtentError`
`missextents.createextent`	run TEX or LATEX immediately to get the requested size
`missextents.createallextent`	run TEX or LATEX immediately to get the hight, width, and depth of the given text at once

`msghandler:` provides a filter for TEX and LATEX messages and defines, which messages are hidden. In the following table the predefined message handlers are described:

msghandler	description
`msghandler.showall`	shows all messages
`msghandler.hideload`	Hides messages which are written when loading packages and including other files. They look like `(file...)` where `file` is a readable file and `...` stands for any text. This message handler is the default handler.
`msghandler.hidegraphicsload`	Hides messages which are written by `includegraphics` of the `graphicx` package. They look like `<file>` where `file` is a readable file.
`msghandler.hidefontwarning`	Hides LATEX font warnings. They look like `LaTeX Font Warning:` and are followed by lines starting with `(Font)`.
`msghandler.hidebuterror`	Hides messages except those with a line which starts with "`!  `".
`msghandler.hideall`	hides all messages

14.3. Constructors

Named parameters of the constructor are used to set global options for the instances of the classes `tex` and `latex`. There are some common options for both classes listed in the following table.

parameter name	default value	description
`defaultmsghandler`	`msghandler.hideload`	default message handler (tuple of message handlers is possible)
`defaultmissextents`	`missextents.returnzero`	default missing extent handler
`texfilename`	`None`	Filename used for running TEX or LATEX. If `None`, a temporary name is used and the files are removed automatically. It can be used to trace errors.

Additionally, the class `tex` has another option described in the following table.

parameter name	default value	description
`lts`	`"10pt"`	Specifies a latex font size file. Those files with the suffix `.lfs` can be created by `createlfs.tex`. Possible values are listed when a requested name couldn't be found.

Instead of the option listed in the table above, for the class `latex` the options described in the following table are available (additionally to the common available options).

parameter name	default value	description
`docclass`	`"article"`	specifies the document class
`docopt`	`None`	specifies options to the document class
`auxfilename`	`None`	Specifies a filename for storing the LATEX `aux` file. This is needed when using labels and references.

14.4. Examples

14.4.1. Example 1

```
from pyx import *

c = canvas.canvas()
t = c.insert(tex.tex())

t.text(0, 0, "Hello, world!")

print "width:", t.textwd("Hello, world!")
print "height:", t.textht("Hello, world!")
print "depth:", t.textdp("Hello, world!")

c.writetofile("tex1")
```

The output of this program is:

```
width: (0.019535 t + 0.000000 u + 0.000000 v + 0.000000 w) m
```

```
height: (0.002441 t + 0.000000 u + 0.000000 v + 0.000000 w) m
depth: (0.000683 t + 0.000000 u + 0.000000 v + 0.000000 w) m
```

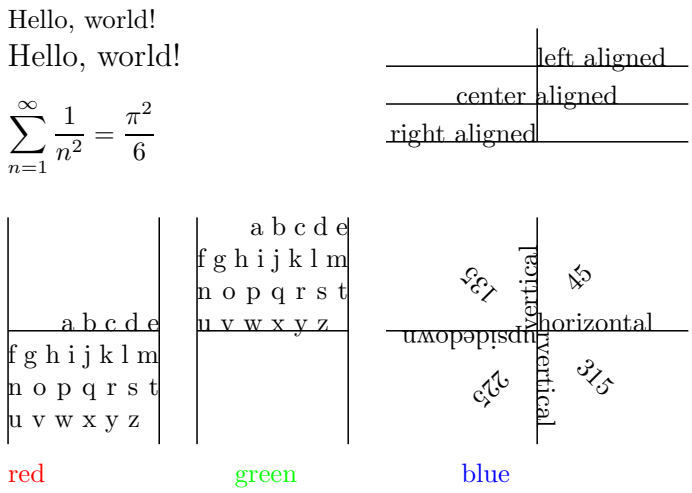
The file `tex1.eps` is created and looks like:

14.4.2. Example 2

```
from pyx import *

c = canvas.canvas()
t = c.insert(tex.tex())

t.text(0, 0, "Hello, world!")
t.text(0, -0.5, "Hello, world!", tex.fontsize.large)
t.text(0, -1.5,
       r"\sum_{n=1}^{\infty} {1\over{n^2}} = {{\pi^2}\over 6}",
       tex.style.math)
c.stroke(path.line(5, -0.5, 9, -0.5))
c.stroke(path.line(5, -1, 9, -1))
c.stroke(path.line(5, -1.5, 9, -1.5))
c.stroke(path.line(7, -1.5, 7, 0))

t.text(7, -0.5, "left aligned") # default is tex.halign.left
t.text(7, -1, "center aligned", tex.halign.center)
t.text(7, -1.5, "right aligned", tex.halign.right)

c.stroke(path.line(0, -4, 2, -4))
c.stroke(path.line(0, -2.5, 0, -5.5))
c.stroke(path.line(2, -2.5, 2, -5.5))

t.text(0, -4,
       "a b c d e f g h i j k l m n o p q r s t u v w x y z",
       tex.valign.top(2))

c.stroke(path.line(2.5, -4, 4.5, -4))
c.stroke(path.line(2.5, -2.5, 2.5, -5.5))
c.stroke(path.line(4.5, -2.5, 4.5, -5.5))

t.text(2.5, -4,
       "a b c d e f g h i j k l m n o p q r s t u v w x y z",
       tex.valign.bottom(2))

c.stroke(path.line(5, -4, 9, -4))
```

```
c.stroke(path.line(7, -5.5, 7, -2.5))

t.text(7, -4, "horizontal")
t.text(7, -4, "vertical", tex.direction.vertical)
t.text(7, -4, "rvertical", tex.direction.rvertical)
t.text(7, -4, "upsidedown", tex.direction.upsidedown)

t.text(7.5, -3.5, "45", tex.direction(45))
t.text(6.5, -3.5, "135", tex.direction(135))
t.text(6.5, -4.5, "225", tex.direction(225))
t.text(7.5, -4.5, "315", tex.direction(315))

t.text(0, -6, "red", color.rgb.red)
t.text(3, -6, "green", color.rgb.green)
t.text(6, -6, "blue", color.rgb.blue)

c.writetofile("tex2")
```

The file `tex2.eps` is created and looks like:



14.5. Known bugs

- The end of the last paragraph in a vertical box (`valign.top` and `valign.bottom`) must be explictly written (by the command `\par` or an empty line) when a paragraph formating parameter is changed locally (like the `\baselineskip` when changing the font size). Otherwise, the information is thrown away due to a closing of the block before the paragraph formatting is performed.
- Due to `dvips` the bounding box is wrong for rotated text. The rotation is just ignored in the bounding box calculation.

- Analysing TEX messages is a difficult subject and the message handlers provided with PYX are not at all perfect in that sense. For the message handlers `msghandler.hideload` and `msghandler.hidegraphicsload` it is known, that they do not correctly handle long filenames splited on several lines by TEX.

14.6. Future of the module tex

While we will certainly keep this module working at least for a while, it is likely that another TEX interface will occure soon. The idea is to get rid of `dvips` and integrate TEX more directly into PYX. The replacement module called `text` becomes available for the first time in PYX 0.3.

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.

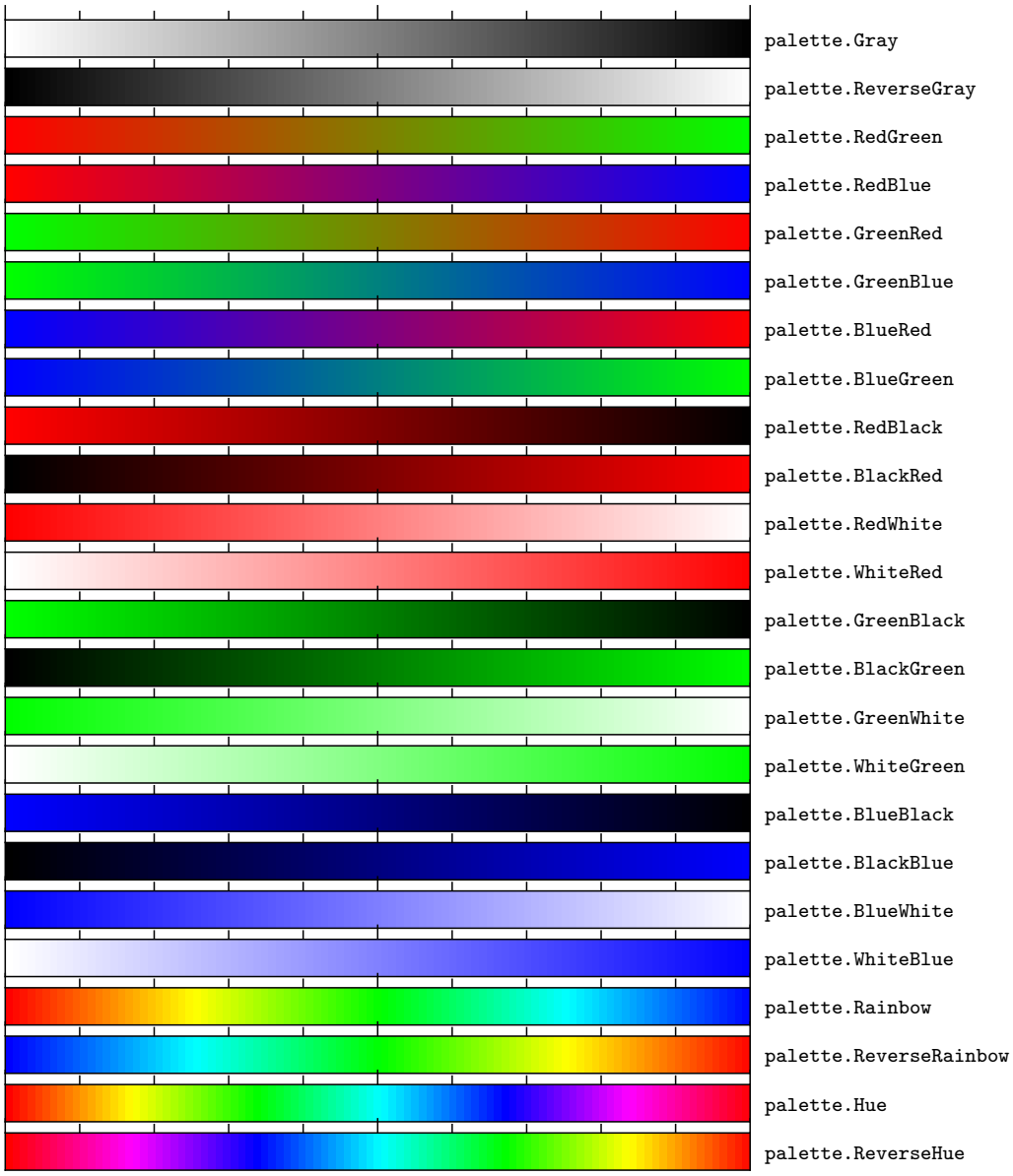
Operators: `+`; `-`; `*`; `/`; `**` and `^` (both for power)

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)

B. Named colors

C. Named palettes



D. Path styles and arrows in canvas module