

<http://pyx.sourceforge.net/>

PyX 0.3

User Manual

Jörg Lehmann <joergl@users.sourceforge.net>

André Wobst <wobsta@users.sourceforge.net>

March 31, 2003

PostScript is a trademark of Adobe Systems Incorporated.

Contents

1	Introduction	6
2	Module unit	7
2.1	Class length	7
2.2	Subclasses of length	8
2.3	Conversion functions	8
3	Module path: PostScript like paths	10
3.1	Class pathel	10
3.2	Class path	11
3.3	Class normpath	13
3.4	Subclasses of path	13
4	Module trafo: linear transformations	14
4.1	Class trafo	14
4.2	Subclasses of trafo	15
5	Module canvas: PostScript interface	16
5.1	Class canvas	16
5.1.1	Basic usage	16
5.1.2	Methods of the class canvas	17
5.2	Patterns	19
5.3	Subclasses of base.PathStyle	19
6	Module text: T_EX/L^AT_EX interface	20
6.1	Basic functionality	20
6.2	The texrunner	20
6.3	T _E X/L ^A T _E X settings	22
6.4	T _E X/L ^A T _E X message parsers	23
6.5	The defaulttexrunner instance	23
7	Module box: convex box handling	24
7.1	polygon	24
7.2	functions working on a box list	25
7.3	rectangular boxes	25

8	Module epsfile: EPS file inclusion	26
9	Module bbox	27
9.1	bbox constructor	27
9.2	bbox methods	28
10	Module color	29
10.1	Color models	29
10.2	Example	29
10.3	Color palettes	30
11	Module data	31
11.1	Reading a table from a file	31
11.2	Accessing columns	32
11.3	Mathematics on columns	32
11.4	Reading data from a sectioned config file	33
11.5	Own datafile readers	33
12	Module graph: graph plotting	34
12.1	Introductory notes	34
12.2	Axes	34
12.2.1	Axes properties	35
12.2.2	Partitioning of axes	35
12.2.3	Painting of axes	39
12.2.4	Linked axes	41
12.2.5	Special purpose axes	41
12.3	Data	43
12.3.1	List of points	43
12.3.2	Functions	44
12.3.3	Parametric functions	44
12.4	Styles	45
12.4.1	Symbols	45
12.4.2	Lines	46
12.4.3	Rectangles	46
12.4.4	Texts	46
12.4.5	Arrows	46
12.4.6	Bars	47
12.4.7	Iterateable style attributes	47
12.5	Keys	48
12.6	X-Y-Graph	48
13	Module tex: T_EX/L^AT_EX interface (obsolete)	50
13.1	Methods	50
13.2	Attributes	51

13.3 Constructors	52
13.4 Examples	53
13.4.1 Example 1	53
13.4.2 Example 2	54
13.5 Known bugs	55
13.6 Future of the module tex	56
A Mathematical expressions	57
B Named colors	58
C Named palettes	59
D Path styles and arrows in canvas module	60

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 <code>length</code>	Type	Unit	Subclass of <code>length</code>	Type	Unit
<code>m(x)</code>	user	m	<code>v_m(x)</code>	visual	m
<code>cm(x)</code>	user	cm	<code>v_cm(x)</code>	visual	cm
<code>mm(x)</code>	user	mm	<code>v_mm(x)</code>	visual	mm
<code>inch(x)</code>	user	inch	<code>v_inch(x)</code>	visual	inch
<code>pt(x)</code>	user	points	<code>v_pt(x)</code>	visual	points
<code>t_m(x)</code>	true	m	<code>w_m(x)</code>	width	m
<code>t_cm(x)</code>	true	cm	<code>w_cm(x)</code>	width	cm
<code>t_mm(x)</code>	true	mm	<code>w_mm(x)</code>	width	mm
<code>t_inch(x)</code>	true	inch	<code>w_inch(x)</code>	width	inch
<code>t_pt(x)</code>	true	points	<code>w_pt(x)</code>	width	points
<code>u_m(x)</code>	user	m			
<code>u_cm(x)</code>	user	cm			
<code>u_mm(x)</code>	user	mm			
<code>u_inch(x)</code>	user	inch			
<code>u_pt(x)</code>	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
<code>to_m(1)</code>	1 in units of m
<code>to_cm(1)</code>	1 in units of cm
<code>to_mm(1)</code>	1 in units of mm
<code>to_inch(1)</code>	1 in units of inch
<code>to_pt(1)</code>	1 in units of points

If `1` 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 `pathels`, namely the four `normpathels` `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 <code>pathel</code>	function
<code>closepath()</code>	closes current subpath
<code>moveto(x, y)</code>	sets current point to (x, y)
<code>rmoveto(dx, dy)</code>	moves current point by (dx, dy)
<code>lineto(x, y)</code>	moves current point to (x, y) while drawing a straight line
<code>rlineto(dx, dy)</code>	moves current point by (dx, dy) while drawing a straight line
<code>arc(x, y, r, angle1, angle2)</code>	appends arc segment in counterclockwise direction with center (x, y) and radius r from <code>angle1</code> to <code>angle2</code> (in degrees).
<code>arcn(x, y, r, angle1, angle2)</code>	appends arc segment in clockwise direction with center (x, y) and radius r from <code>angle1</code> to <code>angle2</code> (in degrees).
<code>arct(x1, y1, x2, y2, r)</code>	appends arc segment of radius r connecting between (x1, y1) and (x2, y2).
<code>rcurveto(dx1, dy1, dx2, dy2, dx3, dy3)</code>	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 necessarily 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 `pathels`. The following simple example generates a triangle: looks like:

```
from pyx import *
from path import *

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

Later on, 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 `pathels` using `p.append(pathel)` to more advanced methods, which are summarized in the following table.

XXX terminology: subpath, ...

path method	function
<code>__init__(*pathels)</code>	construct new <code>path</code> consisting of <code>pathels</code>
<code>append(pathel)</code>	appends <code>pathel</code> to the end of <code>path</code>
<code>arclength(epsilon=1e-5)</code>	returns the total arc length of all <code>path</code> segments in PostScript points with accuracy <code>epsilon</code> . [†]
<code>at(t)</code>	returns the coordinates of the point of <code>path</code> corresponding to the parameter value <code>t</code> . [†]
<code>bbox()</code>	returns the bounding box of the <code>path</code>
<code>begin()</code>	return first point of first subpath of <code>path</code> . [†]
<code>end()</code>	return last point of last subpath of <code>path</code> . [†]
<code>glue(opath)</code>	returns the <code>path</code> glued together with <code>opath</code> , <i>i.e.</i> the last subpath of <code>path</code> and the first one of <code>opath</code> are joined. [†]
<code>intersect(opath, epsilon=1e-5)</code>	returns tuple consisting of two lists of parameter values corresponding to the intersection points of <code>path</code> and <code>opath</code> , respectively. [†]
<code>reversed()</code>	returns the normalized reversed <code>path</code> . [†]
<code>split(t)</code>	returns a tuple consisting of two <code>normpaths</code> corresponding to the <code>path</code> split at the parameter value <code>t</code> . [†]
<code>transformed(trafo)</code>	returns the normalized and accordingly to the linear transformation <code>trafo</code> transformed path. Here, <code>trafo</code> must be an instance of the <code>trafo.trafo</code> 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 `pathels` 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 <code>path</code>	function
<code>line(x1, y1, x2, y2)</code>	a line from the point (x1, y1) to the point (x2, y2)
<code>curve(x0, y0, x1, y1, x2, y2, x3, y3)</code>	a Bézier curve with control points (x0, y0), ..., (x3, y3).
<code>rect(x, y, w, h)</code>	a rectangle with the lower left point (x, y), width w, and height h.
<code>circle(x, y, r)</code>	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` module `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

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

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

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 $A = A_1A_2$ and $\vec{b} = A_1\vec{b}_2 + \vec{b}_1$. 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 $-A^{-1}\vec{b}$.

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

trafo method	function
<code>__init__(matrix=((1,0),(0,1)), vector=(0,0)):</code>	create new trafo instance with transformation matrix and vector .
<code>apply(x, y)</code>	apply trafo to point vector (x,y).
<code>inverse()</code>	returns inverse transformation of trafo .
<code>mirrored(angle)</code>	returns trafo followed by mirroring at line through (0,0) with direction angle in degrees.
<code>rotated(angle, x=None, y=None)</code>	returns trafo followed by rotation by angle degrees around point (x,y), or (0,0), if not given.
<code>scaled(sx, sy=None, x=None, y=None)</code>	returns trafo followed by scaling with scaling factor sx in <i>x</i> -direction, sy in <i>y</i> -direction (sy = sx , if not given) with scaling center (x,y), or (0,0), if not given.
<code>translated(x, y)</code>	returns trafo followed by translation by vector (x,y).
<code>slanted(a, angle=0, x=None, y=None)</code>	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
<code>mirror(angle)</code>	mirroring at line through (0,0) with direction angle in degrees.
<code>rotate(angle, x=None, y=None)</code>	rotation by angle degrees around point (x,y), or (0,0), if not given.
<code>scale(sx, sy=None, x=None, y=None)</code>	scaling with scaling factor sx in <i>x</i> -direction, sy in <i>y</i> -direction (sy = sx , if not given) with scaling center (x,y), or (0,0), if not given.
<code>translate(x, y)</code>	translation by vector (x,y).
<code>slant(a, angle=0, x=None, y=None)</code>	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 `PathStyles` 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 `PathStyles`:

```
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
<code>__init__(*args)</code>	Construct new canvas. <code>args</code> can be instances of <code>trafo.trafo</code> , <code>canvas.clip</code> and/or <code>canvas.PathStyle</code> .
<code>bbox()</code>	Returns the bounding box enclosing all elements of the canvas.
<code>draw(path, *styles)</code>	Generic drawing routine for given <code>path</code> on the canvas (<i>i.e.</i> inserts it together with the necessary <code>newpath</code> command, applying the given <code>styles</code> . Styles can either be instances of <code>base.PathStyle</code> or <code>canvas.PathDeco</code> (or subclasses thereof).
<code>fill(path, *styles)</code>	Fills the given <code>path</code> on the canvas, <i>i.e.</i> inserts it together with the necessary <code>newpath</code> , <code>fill</code> sequence, applying the given <code>styles</code> . Styles can either be instances of <code>base.PathStyle</code> or <code>canvas.PathDeco</code> (or subclasses thereof).
<code>insert(PSOp, *args)</code>	Inserts an instance of <code>base.PSOp</code> into the canvas. If <code>args</code> are present, create a new <code>canvasinstance</code> passing <code>args</code> as arguments and insert it. Returns <code>PSOp</code> .
<code>set(*styles)</code>	Sets the given <code>styles</code> (instances of <code>base.PathStyle</code> or subclasses) for the rest of the canvas.
<code>stroke(path, *styles)</code>	Strokes the given <code>path</code> on the canvas, <i>i.e.</i> inserts it together with the necessary <code>newpath</code> , <code>stroke</code> sequence, applying the given <code>styles</code> . Styles can either be instances of <code>base.PathStyle</code> or <code>canvas.PathDeco</code> (or subclasses thereof).
<code>text(x, y, text, *args)</code>	Inserts <code>text</code> into the canvas (shortcut for <code>insert(texrunner.text(x, y, text, *args))</code>).
<code>texrunner(texrunner)</code>	Sets the <code>texrunner</code> ; default is <code>defaulttexrunner</code> from the <code>text</code> module.
<code>writetofile(filename, paperformat=None, rotated=0, fittosize=0, margin="1 t cm", bbox=None, bboxenlarge="1 t pt")</code>	Writes the canvas to <code>filename</code> . Optionally, a <code>paperformat</code> can be specified, in which case the output will be centered with respect to the corresponding size using the given <code>margin</code> . See <code>canvas._paperformats</code> for a list of known paper formats . Use <code>rotated</code> , if you want to center on a 90° rotated version of the respective paper format. If <code>fittosize</code> 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 <code>bbox</code> manually. In any case, the bounding box becomes enlarged on all side by <code>bboxenlarge</code> . This may be used to compensate for the inability of <code>P_YX</code> 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
<code>painttype</code>	1 (default) for coloured patterns or 2 for uncoloured patterns
<code>tilingtype</code>	1 (default) for constant spacing tilings (patterns are spaced constantly by a multiple of a device pixel), 2 for undistorted 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.
<code>xstep</code>	desired horizontal spacing between pattern cells, use <code>None</code> (default) for automatic calculation from pattern bounding box.
<code>ystep</code>	desired vertical spacing between pattern cells, use <code>None</code> (default) for automatic calculation from pattern bounding box.
<code>bbox</code>	bounding box of pattern. Use <code>None</code> for an automatic determination of the bounding box (including an enlargement by 5 pts on each side.)
<code>trafo</code>	additional transformation applied to pattern or <code>None</code> (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: T_EX/L^AT_EX interface

6.1 Basic functionality

The `text` module seamlessly integrates the famous typesetting technique of T_EX/L^AT_EX into P_X. The basic procedure is:

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

6.2 The `texrunner`

Instances of the class `texrunner` represent a T_EX/L^AT_EX instance. The keyword arguments of the constructor are listed in the following table:

keyword	description
<code>mode</code>	" <code>tex</code> " (default) or " <code>latex</code> "
<code>lfs</code>	Specifies a latex font size file to be used with \TeX . Those files (with the suffix <code>.lfs</code>) can be created by <code>createlfs.tex</code> . Possible values are listed when a requested name could not be found.
<code>docclass</code>	\LaTeX document class; default is " <code>article</code> "
<code>docopt</code>	specifies options for the document class; default is <code>None</code>
<code>usefiles</code> ¹	filenames to be as jobname files for \TeX / \LaTeX ; default: <code>None</code> ; example: (" <code>spam.aux</code> ", " <code>eggs.log</code> ")
<code>waitfortex</code>	wait this number of seconds for a \TeX / \LaTeX response; default 5
<code>texdebug</code>	\TeX / \LaTeX debug messages (boolean); default 0
<code>dvidebug</code>	dvi debug messages like <code>dvitype</code> (boolean); default 0
<code>texmessagestart</code> ^{1,2}	parsers for the \TeX / \LaTeX start message; default: <code>texmessage.start</code>
<code>texmessagedocclass</code> ^{1,2}	parsers for \LaTeX s <code>\documentclass</code> statement; default: <code>texmessage.load</code>
<code>texmessagebegindoc</code> ^{1,2}	parsers for \LaTeX s <code>\begin{document}</code> statement; default: (<code>texmessage.load</code> , <code>texmessage.noaux</code>)
<code>texmessageend</code> ^{1,2}	parsers for \TeX s <code>\end/</code> \LaTeX s <code>\end{document}</code> statement; default: <code>texmessage.texend</code>
<code>texmessagedefaultpreamble</code> ^{1,2}	default parsers for preamble statements; default: <code>texmessage.load</code>
<code>texmessagedefaulttrun</code> ^{1,2}	default parsers for text statements; default: <code>None</code>

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 three methods to be called by the user. The first method is called `set`. It takes the same keyword arguments as the constructor and its purpose is to provide an access to the `texrunners` settings for a given instance. This is important for the `defaulttextrunner`. The `set` method fails, when a modification can't be applied anymore (e.g. \TeX / \LaTeX was already started).

Secondly there is a `preamble` method, which 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.

Last, but not first, there is a `text` method. The first two parameters are the x, y position of the output to be generated. The third parameter is a \TeX / \LaTeX expression and further



Figure 6.1: valign example

parameters are attributes for this command. Those attributes might be $\text{T}_{\text{E}}\text{X}/\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ settings as described below, $\text{T}_{\text{E}}\text{X}/\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ message parsers as described below as well, $\text{P}_{\text{X}}\text{X}$ transformations, and $\text{P}_{\text{X}}\text{X}$ 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.

6.3 $\text{T}_{\text{E}}\text{X}/\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ 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, $\text{T}_{\text{E}}\text{X}/\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ expressions are handled in horizontal mode (so-called LR-mode in $\text{T}_{\text{E}}\text{X}/\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$; 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 $\text{T}_{\text{E}}\text{X}/\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$).

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 $\text{T}_{\text{E}}\text{X}/\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$

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

`size.Huge`

6.4 T_EX/L_AT_EX message parsers

Message parsers are used to scan the output of T_EX/L_AT_EX. 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
<code>texmessage.load</code>	loading of files (accept <code>(file ...)</code>)
<code>texmessage.graphicsload</code>	loading of graphic files (accept <code><file ...></code>)
<code>texmessage.ignore</code>	accept everything as a valid output

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

6.5 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

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

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

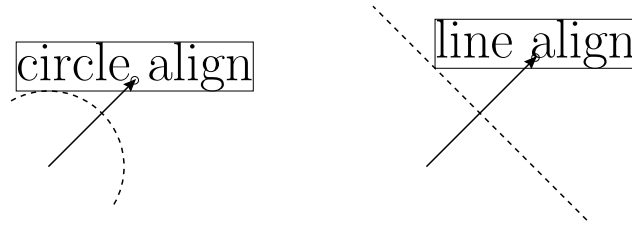


Figure 7.1: circle and line alignment examples (equal direction and distance)

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

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

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

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

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

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

9 Module `bbox`

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

9.1 `bbox` constructor

The `bbox` constructor accepts the following keyword arguments

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

9.2 bbox methods

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

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

10 Module color

10.1 Color models

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

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

10.2 Example

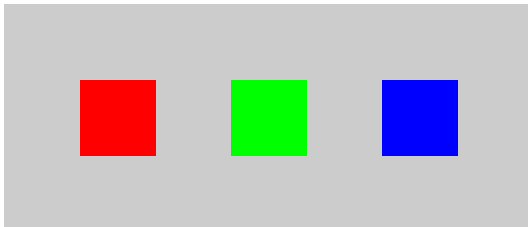
```
from pyx import *

c = canvas.canvas()

c.fill(path.rect(0, 0, 7, 3), color.gray(0.8))
c.fill(path.rect(1, 1, 1, 1), color.rgb.red)
c.fill(path.rect(3, 1, 1, 1), color.rgb.green)
c.fill(path.rect(5, 1, 1, 1), color.rgb.blue)

c.writetofile("color")
```

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



10.3 Color palettes

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

A list of named palettes is available in appendix [C](#).

11 Module data

11.1 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
<code>commentpattern</code>	start a comment line; default: <code>re.compile(r"#+ !+ %+)\s*")</code>
<code>stringpattern</code>	a string column; default: <code>re.compile(r"\"(.*)\"(\s+ \$)")</code>
<code>columnpattern</code>	any other column; default: <code>re.compile(r"(.*)\"(\s+ \$)")</code>
<code>skiphead</code>	skip first data lines; default: 0
<code>skiptail</code>	skip last data lines; default: 0
<code>every</code>	only take every <code>every</code> 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 trailing 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.

11.2 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 occurs, 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.

11.3 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`.

11.4 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).

11.5 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 list of data points as the `data` keyword argument. A data point itself is a list of floats and/or strings. Additionally a title list can be given in the `titles` argument.

12 Module graph: graph plotting

12.1 Introductory notes

The graph module is considered to be in constant, gradual development. For the moment we concentrate ourselves 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 whole 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 independent as it is useful and possible in order to make them reusable for different graph types. They are also replaceable by the user to get more specialized graph drawing tasks done without needing to implement a whole 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. By that, all other components can be reused for different graph geometries.

12.2 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
<code>linaxis</code>	linear axis
<code>logaxis</code>	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.

12.2.1 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
<code>title</code>	axis title
<code>min</code>	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
<code>max</code>	as above, but for the maximum
<code>reverse</code>	boolean; exchange minimum and maximum (might be used without setting minimum and maximum); if <code>min</code> / <code>max</code> and <code>reverse</code> is set, they cancel each other
<code>divisor</code>	numerical divisor for the axis partitioning; default: 1
<code>suffix</code>	a suffix to indicate the divisor within an automatic axis labeling
<code>datavmin</code>	minimal graph coordinate when adjusting the axis minima to the graph data; default: 0.05 (or 0, when <code>min</code> is present)
<code>datavmax</code>	as above, but for the maximum; default: 0.95 (or 1, when <code>max</code> is present)
<code>tickvmin</code>	minimal graph coordinate for placing ticks to the axis; default: 0
<code>tickvmax</code>	as above, but for the maximum; default: 1
<code>part</code>	axis partitioning (described below)
<code>rate</code>	axis partition rating (described below)
<code>dense</code>	dense parameter for the axis partition rating; if not set, the dense value for the graph is used
<code>painter</code>	axis painter (described below)

12.2.2 Partitioning of axes

The definition of ticks and labels appropriate to an axis range is called partitioning. The axis partitioning within `PYX` uses rational arithmetics, which avoids any kind of rounding problems to the cost of performance. The class `frac` supplies a rational number. 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` and `labellevel`. When those values are `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 `text` 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.

We will first discuss manual partitioning schemes, namely a plain manual partition, another appropriate for linear axes and a third one for logarithmic axes. These partition schemes might be used directly or indirectly via automatic axis partitioning schemes.

Manual partitioning

The class `manualpart` creates a manual partition where every single tick, label etc. is set independently from each other as described by named parameters of the constructor:

argument name	default	description
<code>ticks</code>	<code>None</code>	position of ticks, subticks, etc. (see below)
<code>labels</code>	<code>None</code>	position of labels, sublabels, etc. (see below)
<code>texts</code>	<code>None</code>	force text at labels, sublabels, etc. (see below)
<code>mix</code>	<code>()</code>	ordered tick list to be merged into the result

The parameters `ticks`, `labels`, and `texts` can either be a sequence, or a sequence of sequences. (When it is not a sequence at all, it is converted to a sequence with a single entry.) When it is a sequence of sequences, than the first sequence stands for the ticks, labels, and texts of the labels, the second sequence stands for the subticks, sublabels, and texts of the sublabels, and so on. When it is just a sequence, it stands for the ticks, labels and texts of the labels and no subticks, sublabels and subtexts will be created.

The single entries of `ticks` and `labels` can either be a `frac` or a string, which will be converted to a `frac`. However, a float is not valid in order to avoid a conversion from a float to a `frac`. Valid strings are just numbers like "0.1", or fractions like "1/10".

Partitioning of linear axes

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

argument name	default	description
<code>ticks</code>	<code>None</code>	distance between ticks, subticks, etc. (see comment below); when the parameter is <code>None</code> , ticks will get placed at labels
<code>labels</code>	<code>None</code>	distance between labels, sublabels, etc. (see comment below); when the parameter is <code>None</code> , labels will get placed at ticks
<code>extendtick</code>	<code>0</code>	allow for a range extension to include the next tick of the given level
<code>extendlabel</code>	<code>None</code>	as above, but for labels
<code>epsilon</code>	<code>1e-10</code>	allow for exceeding the range by that relative value
<code>texts</code>	<code>None</code>	as in <code>manualpart</code>
<code>mix</code>	<code>()</code>	as in <code>manualpart</code>

The `ticks` and `labels` can either be a sequence or just a single entry. When a sequence is provided, the first entry stands for the tick or label, respectively, the second for the subtick or sublabel, and so on. The entries can either be a `frac` or a string, as in `manualpart`.

Partitioning of logarithmic axes

The class `logpart` create a logarithmic partition. The class has the same arguments as `linpart` upto the interpretation of two arguments `ticks` and `labels`. Both parameters can contain just a single entry or a sequence — the interpretation of those possibilities is the same as it was for `linpart`. The entries have to be `shiftfracs`, which contains a `frac` for the shift, say s , and a list of `frac` for the positions, say p_i . Valid positions are then $s^n p_i$, where n can be any integer number. Within `logpart` there are numerous predefined `shiftfracs`, namely:

name	values it describes
<code>shift5fracs1</code>	1 and multiple of 10^5
<code>shift4fracs1</code>	1 and multiple of 10^4
<code>shift3fracs1</code>	1 and multiple of 10^3
<code>shift2fracs1</code>	1 and multiple of 10^2
<code>shiftfracs1</code>	1 and multiple of 10
<code>shiftfracs125</code>	1, 2, 5 and multiple of 10
<code>shiftfracs1to9</code>	1, 2, ..., 9 and multiple of 10

Automatic partitioning of linear axes

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

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

The default value for the argument `list`, namely `defaultlist`, 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. Additionally, as most likely already from the default value of the argument `list`, the given fractions are automatically multiplied or divided by 10 in order to fit better to the axis range. Therefore these additional partitioning possibilities must not be given explicitly.

Automatic partitioning of logarithmic axes

When no explicit axis partitioning is given in the constructor 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
<code>list</code>	<code>defaultlist</code>	list of pairs with possible values for the ticks and labels parameters of <code>logpart</code>
<code>extendtick</code>	<code>0</code>	allow for a range extension to include the next tick of the given level
<code>extendlabel</code>	<code>None</code>	as above, but for labels
<code>epsilon</code>	<code>1e-10</code>	allow for exceeding the range by that relative value
<code>mix</code>	<code>()</code>	as in <code>manualpart</code>

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

```
((shiftfrac1, shiftfrac1to9), # ticks & subticks,
 (shiftfrac1, shiftfrac125)), # labels & sublevels
((shiftfrac1, shiftfrac1to9), None), # ticks & subticks, labels=ticks
((shift2frac1, shiftfrac1), None), # ticks & subticks, labels=ticks
((shift3frac1, shiftfrac1), None), # ticks & subticks, labels=ticks
((shift4frac1, shiftfrac1), None), # ticks & subticks, labels=ticks
((shift5frac1, shiftfrac1), 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 `rate` 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 whole 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 **dense** of the axis (or graph) constructor. The rating is adjusted to the default density 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. This parameter was taken out of the rating class for easier access.

12.2.3 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
<code>innerticklengths</code> ^{1,4}	tick length of inner ticks (visual length); default: <code>axispainter.defaultticklengths</code>
<code>outerticklengths</code> ^{1,4}	as before, but for outer ticks; default: <code>None</code>
<code>tickattrs</code> ^{2,4}	stroke attributes for ticks; default: <code>()</code>
<code>gridattrs</code> ^{2,4}	stroke attributes for grid lines; default: <code>None</code>
<code>zerolineattrs</code> ^{3,4}	stroke attributes for a grid line at axis value 0; default: <code>()</code>
<code>baselineattrs</code> ^{3,4}	stroke attributes for the axis baseline; default: <code>canvas.linecap.square</code>
<code>labeldist</code>	label distance from axis (visual length); default: <code>"0.3 cm"</code>
<code>labelattrs</code> ^{2,4}	text attributes for labels; default: <code>((textmodule.halign.center,</code> <code>textmodule.vshift.mathaxis),</code> <code>(textmodule.size.footnotesize,</code> <code>textmodule.halign.center,</code> <code>textmodule.vshift.mathaxis))</code>
<code>labeldirection</code> ⁴	relative label direction (see below); default: <code>None</code>
<code>labelhequalize</code>	set width of labels to its maximum (boolean); default: <code>0</code>
<code>labelvequalize</code>	set height and depth of labels to their maxima (boolean); default: <code>1</code>
<code>titledist</code>	title distance from labels (visual length); default: <code>"0.3 cm"</code>
<code>titleattrs</code> ^{3,4}	text attributes for title; default: <code>()</code>
<code>titledirection</code> ⁴	relative title direction (see below); default: <code>axispainter.paralleltext</code>
<code>titlepos</code>	title position in graph coordinates; default: <code>0.5</code>
<code>fractype</code>	text creation for labels (see below); default: <code>axispainter.fractypeauto</code>
<code>ratfracsuffixenum</code>	write suffix at the enumerator (boolean); default: <code>1</code>
<code>ratfraccover</code>	text for fraction line; default: <code>r"\over"</code>
<code>decfracpoint</code>	decimal point; default: <code>."</code>
<code>decfracequal</code>	equalize number of decimal digits (boolean); default: <code>0</code>
<code>expfractimes</code>	text between factor and decimal power; default: <code>r"\cdot"</code>
<code>expfracpre1</code>	allow factor 1 before a decimal power (boolean); default: <code>0</code>
<code>expfracminexp</code>	minimal exponent for decimal power; default: <code>4</code>
<code>suffix0</code>	when a suffix is <code>x</code> write <code>0x</code> instead of <code>0</code> (boolean); default: <code>0</code>
<code>suffix1</code>	when a suffix is <code>x</code> write <code>1x</code> instead of <code>x</code> (boolean); default: <code>0</code>

¹ The parameter should be a sequence, where the entries are attributes for the different levels. When the level is larger then the sequence length, `None` is assumed. When the parameter is not a sequence, it is applied to all levels.

² The parameter should be a sequence of sequences, where the entries are attributes for the different levels. When the level is larger then the sequence length, `None` is assumed. When the parameter is not a sequence of sequences, it is applied to all levels.

³ The parameter should be a sequence. When the parameter is not a sequence, the

parameter is interpreted as a sequence with a single entry.

⁴ The feature can be turned off by the value `None`. Within sequences or sequences of sequences, the value `None` might be used to turn off the feature for some levels selectively.

Relative directions for labels (`labeldirection`) and titles (`titledirection`) are basically a float number in degree. The text direction is calculated relatively to the baseline of the axis and is added as an attribute of the text, when no direction was already provided. The relative direction prevents upside down text by flipping it by 180 degrees. For convenience, the two self-explanatory values `axispainter.parallelttext` and `axispainter.orthogonaltext` are available.

The `fractype` parameter determines the creation of label texts. There are three types available, which can be forced by providing them to the `fractype` parameter. The possibilities are listed in the following table.

<code>fractype</code>	description	example
<code>axispainter.fractypedec</code>	decimal	0.1
<code>axispainter.fractypeexp</code>	decimal with exponent	$2 \cdot 10^4$
<code>axispainter.fractyperat</code>	rational	$\frac{1}{2}$
<code>axispainter.fractypeauto</code>	automatic (see below)	

For the default `axispainter.fractypeauto` the three possibilities are selected depending on some simple rules: `axispainter.fractyperat` is used, when the axis provides a suffix, `axispainter.fractypeexp` is used, when the exponent exceed `expfracminexp`, and `axispainter.fractypedec` is used otherwise.

12.2.4 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. The constructor of `linkaxis` receives the axis it is linked to as its first parameter. Additionally, the named parameter `title` contains an axis title (default is `None`) and the named parameter `painter` refers to an `axispainter` (default is `linkaxispainter`). This `linkedaxispainter` is a slightly modified version of the standard `axispainter`. Hence it can receive all the parameters as the `axispainter` and only the default value of the parameter `zerolineattr` is changed to `None` compared to the `axispainter` previously discussed. Additionally, two parameters are added, namely `skipticklevel` and `skiplabellevel`. They are used to build the tick list to be plotted at the linked axis. Ticks and labels at levels equal or higher as the provided values get ignored. The default is `None` (do not ignore any ticks) for the ticks and 0 (ignore all labels) for the labels.

12.2.5 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)
<code>splitlist</code>	a single number or a list split points of the positions of the axis breaks in graph coordinates; the value <code>None</code> forces <code>relsizesplitdist</code> to be used; default: 0.5
<code>splitdist</code>	gap of the axis break; default: 0.1
<code>relsizesplitdist</code>	used when <code>splitlist</code> entries are <code>None</code> ; 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
<code>title</code>	axis title
<code>painter</code>	axis painter; default: <code>splitaxispainter()</code> (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
<code>breaklinesdist</code>	(visual) distance between the break lines; default: 0.05
<code>breaklineslength</code>	(visual) length of break lines; default: 0.5
<code>breaklinesangle</code>	angle of the breakline with respect to the axis; default: -60
<code>breaklinesattrs</code>	stroke attributes for the break lines (<code>None</code> 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`. 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: <code>None</code>
multisubaxis	allow for multiple subaxis (boolean); default: 0
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: <code>0.5*dist</code>
lastdist	as before but for the last bar
names	tuple of name identifiers for bars; when set, no other identifiers are allowed; default: <code>None</code>
texts	dictionary translating names into label texts (otherwise just the names are used); default: <code>{}</code>
painter	axis painter; default: <code>baraxispainter</code> (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 receives. However, the names can be forced and fixed.

Bar axes can be recursive. Thus for a given value, an appropriate subaxis is chosen (usually another bar axis). Usually only a single subaxis is needed, because for each value the same recursive subaxis transformation has to be applied. However, this prevents the subaxis to be painted, but as soon as `multisubaxis` is turned on, 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 (right now, this is only available for the bar axis). 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 `itledist`, `titleattrs`, `itledirection`, `titlepos`, and `baselineattrs`. Furthermore the parameters `innerticklength` and `outerticklength` work like their counterparts in the `axispainter`, but only plain values are allowed there (no tuples). However, they are both `None` by default and no ticks get plotted. Then there is a 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.

12.3 Data

12.3.1 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 single 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 11; 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.

12.3.2 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
<code>min</code>	<code>None</code>	minimal value for the variable parameter; when <code>None</code> , the axis data range will be used
<code>max</code>	<code>None</code>	as above, but for the maximum
<code>points</code>	<code>100</code>	number of points to be calculated
<code>parser</code>	<code>mathtree.parser()</code>	parser for the mathematical expression
<code>context</code>	<code>None</code>	dictionary of extern variables and functions
<code>title</code>	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 rapidly changing functions, handling of divergencies) are likely to be added in future releases.

12.3.3 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`.

12.4 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.

12.4.1 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
<code>symbol</code>	<code>changesymbol.cross()</code>	symbol to be used (see below)
<code>size</code>	<code>"0.2 cm"</code>	size of the symbol (visual length)
<code>symbolattrs</code>	<code>canvas.stroked()</code>	draw attributes for the symbol
<code>errorscale</code>	<code>0.5</code>	size of the errorbar caps (relative to the symbol size)
<code>errorbarattrs</code>	<code>()</code>	stroke attributes for the errorbars
<code>lineattrs</code>	<code>None</code>	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 12.4.7 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
<code>X</code>	position of the symbol
<code>Xmin</code>	minimum for the errorbar
<code>Xmax</code>	maximum for the errorbar
<code>dX</code>	relative size of the errorbar: <code>Xmin, Xmax = X-dX, X+dX</code>
<code>dXmin</code>	relative minimum <code>Xmin = X-dXmin</code>
<code>dXmax</code>	relative maximum <code>Xmax = X+dXmax</code>

12.4.2 Lines

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

12.4.3 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.

12.4.4 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 sequence 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`.

12.4.5 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
<code>linelength</code>	"0.2 cm"	length of a the arrow line (visual length)
<code>arrowattrs</code>	()	stroke attributes
<code>arrowsize</code>	"0.1 cm"	size of the arrow (visual length)
<code>arrowdict</code>	{}	attributes to be used in the <code>earrow</code> constructor
<code>epsilon</code>	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).

12.4.6 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
<code>fromzero</code>	bars start at zero (boolean); default: 1
<code>stacked</code>	stack bars (boolean/integer); for values bigger than 1 it is the number of bars to be stacked; default: 0
<code>skipmissing</code>	skip entries in the bar axis, when datapoints are missing; default: 1
<code>xbar</code>	bars parallel to the graphs x-direction (boolean); default: 0
<code>barattrs</code>	fill attributes; default: (<code>canvas.stroked(color.gray.black)</code> , <code>changecolor.Rainbow()</code>)

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

12.4.7 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 sequence 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
<code>changelinestyle</code>	iterates linestyles solid, dashed, dotted, dashedotted
<code>changestrokedfilled</code>	iterates (<code>canvas.stroked()</code> , <code>canvas.filled()</code>)
<code>changefilledstroked</code>	iterates (<code>canvas.filled()</code> , <code>canvas.stroked()</code>)

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 sequence 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 intended to be used in combination with `changestrokedfilled` and `changefilledstroked`.

12.5 Keys

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

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

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

12.6 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 12.2. 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
<code>xpos</code>	"0"	x position of the graph (user length)
<code>ypos</code>	"0"	y position of the graph (user length)
<code>width</code>	None	width of the graph area (axes are outside of that range)
<code>height</code>	None	as abovem, but for the height
<code>ratio</code>	goldenrule	width/height ratio when only a width or height is provided
<code>backgroundattrs</code>	None	background attributes for the graph area
<code>axisdist</code>	"0.8 cm"	distance between axis (visual length)
<code>key</code>	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 12.3. 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 drawn 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 functionality 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.

13 Module tex: T_EX/L^AT_EX interface (obsolete)

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

13.1 Methods

Text in P_YX is created by T_EX or L^AT_EX. 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 T_EX or L^AT_EX followed by `dvips` to create the requested text. T_EX is used by instances of the class `tex` while L^AT_EX is used by `latex`. Up to the constructor and the advanced possibilities in L^AT_EX 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 <code>*attr</code>
<code>text(x, y, cmd, *attr)</code>	print cmd	<code>style</code> , <code>fontsize</code> , <code>halign</code> , <code>valign</code> , <code>direction</code> , <code>color</code> , <code>msghandler(s)</code>
<code>define(cmd, *attr)</code>	execute cmd	<code>msghandler(s)</code>
<code>textwd(cmd, *attr)</code>	width of cmd	<code>style</code> , <code>fontsize</code> , <code>misextents</code> , <code>msghandler(s)</code>
<code>textht(cmd, *attr)</code>	height of cmd	<code>style</code> , <code>fontsize</code> , <code>valign</code> , <code>misextents</code> , <code>msghandler(s)</code>
<code>textdp(cmd, *attr)</code>	depth of cmd	<code>style</code> , <code>fontsize</code> , <code>valign</code> , <code>misextents</code> , <code>msghandler(s)</code>

There are some common rules:

- `cmd` stands for a T_EX or L^AT_EX 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 L^AT_EX 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 P_YX 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 `misextents` attribute.

- All commands are passed to $\text{T}_{\text{E}}\text{X}$ or $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ 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 $\text{T}_{\text{E}}\text{X}$ or $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ and `dvips` is just inserted.
- The trailing `*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 occur only once.

13.2 Attributes

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

`fontsize`: specifies the $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ 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 $\text{T}_{\text{E}}\text{X}$ dimension `\linewidth`.

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

`color`: stands for any P_{X} color (see section 10), default is `color.gray.black`

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

misextents	description
<code>misextents.returnzero</code>	returns zero (default)
<code>misextents.returnzeroquiet</code>	as above, but does not return a warning via <code>atexit</code>
<code>misextents.raiseerror</code>	raise <code>TexMissExtentError</code>
<code>misextents.createextent</code>	run <code>TeX</code> or <code>LaTeX</code> immediately to get the requested size
<code>misextents.createallextent</code>	run <code>TeX</code> or <code>LaTeX</code> 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
<code>msghandler.showall</code>	shows all messages
<code>msghandler.hideload</code>	Hides messages which are written when loading packages and including other files. They look like <code>(file...)</code> where <code>file</code> is a readable file and <code>...</code> stands for any text. This message handler is the default handler.
<code>msghandler.hidegraphicsload</code>	Hides messages which are written by <code>includegraphics</code> of the <code>graphicx</code> package. They look like <code><file></code> where <code>file</code> is a readable file.
<code>msghandler.hidefontwarning</code>	Hides <code>LaTeX</code> font warnings. They look like <code>LaTeX Font Warning:</code> and are followed by lines starting with <code>(Font)</code> .
<code>msghandler.hidebuterror</code>	Hides messages except those with a line which starts with <code>“! ”</code> .
<code>msghandler.hideall</code>	hides all messages

13.3 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
<code>defaultmsghandler</code>	<code>msghandler.hideload</code>	default message handler (tuple of message handlers is possible)
<code>defaultmissextexts</code>	<code>missextexts.returnzero</code>	default missing extent handler
<code>texfilename</code>	<code>None</code>	Filename used for running <code>TEX</code> or <code>LATEX</code> . If <code>None</code> , 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
<code>lts</code>	<code>"10pt"</code>	Specifies a latex font size file. Those files with the suffix <code>.lfs</code> can be created by <code>createlfs.tex</code> . 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
<code>docclass</code>	<code>"article"</code>	specifies the document class
<code>docopt</code>	<code>None</code>	specifies options to the document class
<code>auxfilename</code>	<code>None</code>	Specifies a filename for storing the <code>LATEX</code> aux file. This is needed when using labels and references.

13.4 Examples

13.4.1 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:

Hello, world!

13.4.2 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, "upsideup", tex.direction.upsideup)

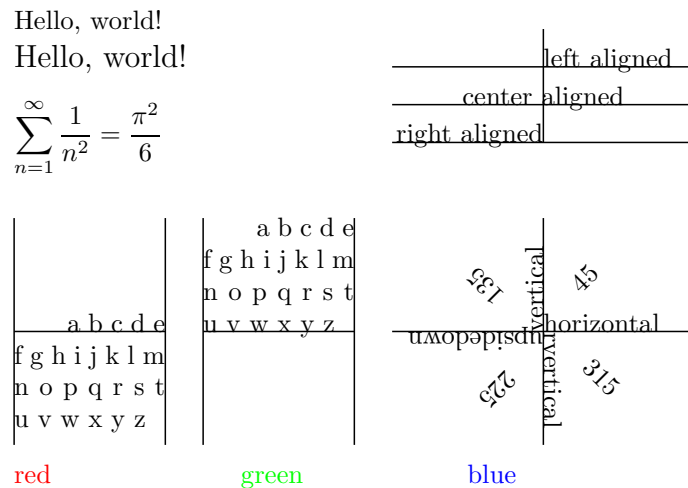
t.text(7.5, -3.5, "45", tex.direction(45))
t.text(6.5, -3.5, "135", tex.direction(135))
t.text(6.5, -4.5, "225", tex.direction(225))
t.text(7.5, -4.5, "315", tex.direction(315))

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:



13.5 Known bugs

- The end of the last paragraph in a vertical box (`valign.top` and `valign.bottom`) must be explicitly written (by the command `\par` or an empty line) when a paragraph formatting 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 .

13.6 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 RYX mathematical expressions can be provided in form of string parameters. They are evaluated by the module `mathtree`. This module is not described further 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)

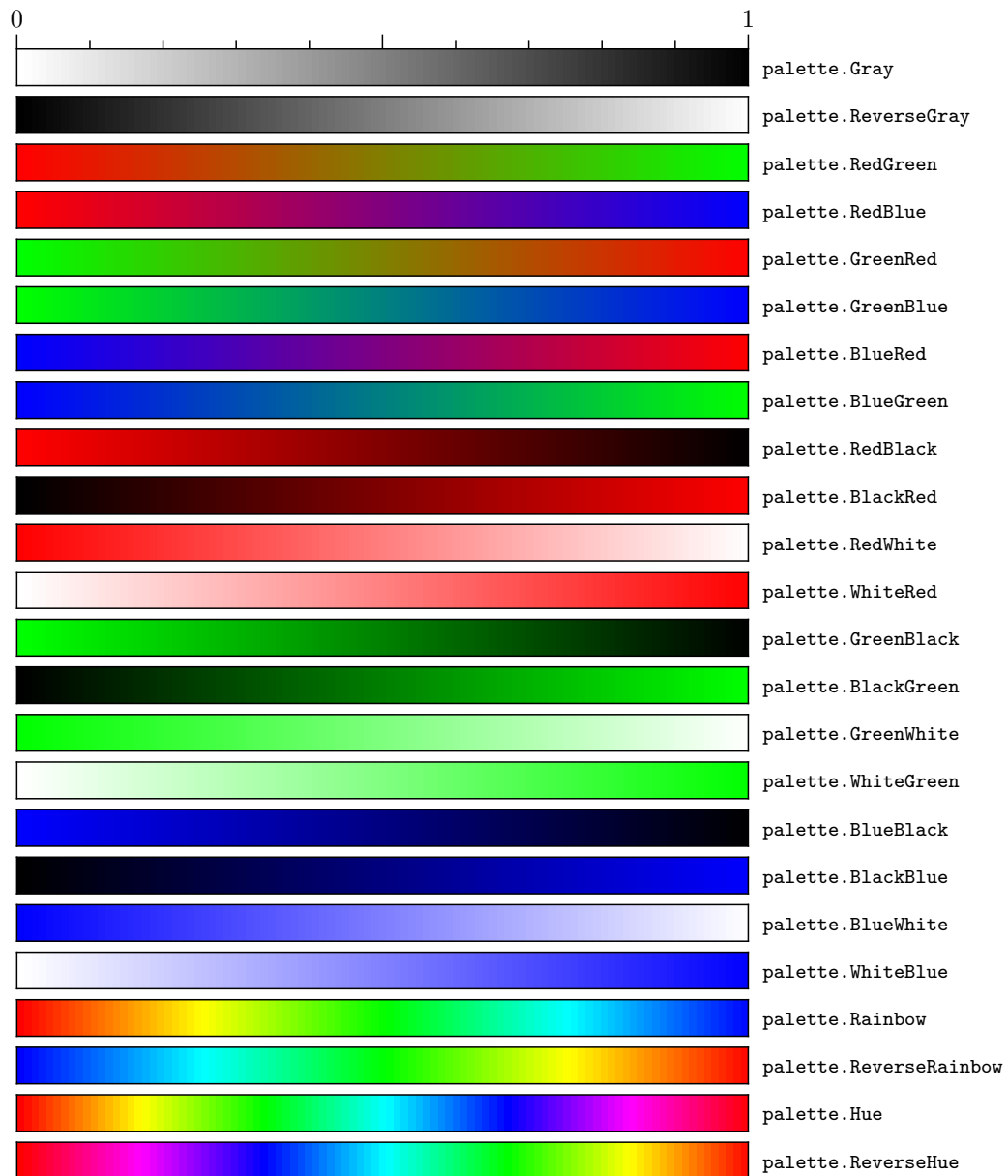
Functions: `neg` (negate); `abs` (absolute value); `sgn` (signum); `sqrt` (square root); `exp`; `log` (natural logarithm); `sin`, `cos`, `tan`, `asin`, `acos`, `atan` (trigonometric functions in radian units); `sind`, `cosd`, `tand`, `asind`, `acosd`, `atand` (as before but in degree units); `norm` ($\sqrt{a^2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

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









































B Named colors

	grey.black		cmyk.WildStrawberry		cmyk.Cerulean
	grey.white		cmyk.Salmon		cmyk.Cyan
			cmyk.CarnationPink		cmyk.ProcessBlue
	rgb.red		cmyk.Magenta		cmyk.SkyBlue
	rgb.green		cmyk.VioletRed		cmyk.Turquoise
	rgb.blue		cmyk.Rhodamine		cmyk.TealBlue
			cmyk.Mulberry		cmyk.Aquamarine
	cmyk.GreenYellow		cmyk.RedViolet		cmyk.BlueGreen
	cmyk.Yellow		cmyk.Fuchsia		cmyk.Emerald
	cmyk.Goldenrod		cmyk.Lavender		cmyk.JungleGreen
	cmyk.Dandelion		cmyk.Thistle		cmyk.SeaGreen
	cmyk.Apricot		cmyk.Orchid		cmyk.Green
	cmyk.Peach		cmyk.DarkOrchid		cmyk.ForestGreen
	cmyk.Melon		cmyk.Purple		cmyk.PineGreen
	cmyk.YellowOrange		cmyk.Plum		cmyk.LimeGreen
	cmyk.Orange		cmyk.Violet		cmyk.YellowGreen
	cmyk.BurntOrange		cmyk.RoyalPurple		cmyk.SpringGreen
	cmyk.Bittersweet		cmyk.BlueViolet		cmyk.OliveGreen
	cmyk.RedOrange		cmyk.Periwinkle		cmyk.RawSienna
	cmyk.Mahogany		cmyk.CadetBlue		cmyk.Sepia
	cmyk.Maroon		cmyk.CornflowerBlue		cmyk.Brown
	cmyk.BrickRed		cmyk.MidnightBlue		cmyk.Tan
	cmyk.Red		cmyk.NavyBlue		cmyk.Gray
	cmyk.OrangeRed		cmyk.RoyalBlue		cmyk.Black
	cmyk.RubineRed		cmyk.Blue		cmyk.White

C Named palettes



D Path styles and arrows in canvas module

	<code>linecap.butt</code> (default)		<code>miterlimit.less than 180deg</code>
	<code>linecap.round</code>		<code>miterlimit.less than 90deg</code>
	<code>linecap.square</code>		<code>miterlimit.less than 60deg</code>
			<code>miterlimit.less than 45deg</code>
	<code>linejoin.miter</code> (default)		<code>miterlimit.less than 11deg</code> (default)
	<code>linejoin.round</code>		
	<code>linejoin.bevel</code>		<code>dash((1, 1, 2, 2, 3, 3), 0)</code>
			<code>dash((1, 1, 2, 2, 3, 3), 1)</code>
	<code>linestyle.solid</code> (default)		<code>dash((1, 2, 3), 2)</code>
	<code>linestyle.dashed</code>		<code>dash((1, 2, 3), 3)</code>
	<code>linestyle.dotted</code>		<code>dash((1, 2, 3), 4)</code>
	<code>linestyle.dashdotted</code>		
			<code>earrow.Small</code>
	<code>linewidth.THIN</code>		<code>earrow.Small</code>
	<code>linewidth.THIn</code>		<code>earrow.small</code>
	<code>linewidth.Thin</code>		<code>earrow.normal</code>
	<code>linewidth.Thin</code>		<code>earrow.large</code>
	<code>linewidth.thin</code>		<code>earrow.Large</code>
	<code>linewidth.normal</code> (default)		<code>earrow.LArge</code>
	<code>linewidth.thick</code>		
	<code>linewidth.Thick</code>		<code>barrow.normal</code>
	<code>linewidth.Thick</code>		
	<code>linewidth.THICK</code>		
	<code>linewidth.THICK</code>		
	<code>linewidth.THICK</code>		