http://pyx.sourceforge.net/

PyX 0.1
User Manual

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

January 17, 2003

Contents

1. Introduction

PyX is a python package to create encapsulated PostScript figures. It provides classes and methods to access basic PostScript functionality at an abstract level. At the same time the emerging structures are very convenient to produce all kinds of drawings in a non-interactive way. In combination with the python language itself the user can just code any complexity of the figure wanted. Additionally an TEX/LATEX interface enables one to use the famous high quality typesetting within the figures.
A major part of PYX on top of the already described basis is the provision of high level functionality for complex tasks like 2d plots in publication-ready quality.

2. Module unit

With the `unit` module PyX makes available classes and functions for the specification and manipulation of lengths. As usual, lengths consist of a number together with a measurement unit, *e.g.* 1 cm, 50 points, 0.42 inch. In addition, lengths in PyX are composed of the four types "true", "user", "visual" and "width", *e.g.* 1 user cm, 50 true points, (0.42 visual + 0.2 width) inch. As their name tells, 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 are differ with respect to the type of object they are applied:

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

visual length: used for sizes of visual elements, like arrows, text, etc.

width length: used for line widths

For instance, if you just 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 `unit.default_unit`.
- For `length(string)` the `string` has to consist of a maximum of three parts separated by one or more whitespaces:

  quantifier: integer/float value

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

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

Note that `default_unit` is initially set to `"cm"`, but can be changed at any time by the user. For instance, use

```
unit.default_unit = "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, which accepts the name arguments `uscale`, `vscale`, and `wscale`. For example, if you like to change the thickness of all lines 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 length 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`.

Subclass of `length`	Type	Unit	Subclass of `length`	Type	Unit
`m(x)`	user	m	`t_m(x)`	true	m
`cm(x)`	user	cm	`t_cm(x)`	true	cm
`mm(x)`	user	mm	`t_mm(x)`	true	mm
`inch(x)`	user	inch	`t_inch(x)`	true	inch
`pt(x)`	user	points	`t_pt(x)`	true	points

Here, `x` is either a number or a string.

2.3. Conversion functions

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

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

If `l` is not yet a `length` instance, it is converted first to a such, 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 help of the path module it is possible to construct PostScript like paths, which are one of the main building blocks for the generation of drawings. To that end it provides

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

3.1. Class pathel

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

Subclass of `pathel`	function
`closepath()`	closes current subpath
`moveto(x, y)`	sets current point to (`x`, `y`)
`rmoveto(dx, dy)`	moves current point relative by (`dx`, `dy`)
`lineto(x, y)`	appends straight line from current point to (`x`, `y`)
`rlineto(dx, dy)`	appends straight line from current point relative by (`dx`, `dy`)

`arc(x, y, r,`
`angle1, angle2)`
	appends arc segment in counterclockwise direction with center (`x`, `y`) and radius `r` from `angle1` to `angle2` (in degrees).

`arcn(x, y, r,`
`angle1, angle2)`
	appends arc segment in clockwise direction with center (`x`, `y`) and radius `r` from `angle1` to `angle2` (in degrees).

`arct(x1, y1, x2, y2, r)`	appends arc segment with radius `r` which connects between (`x1`, `y1`) and (`x2`, `y2`).

`rcurveto(dx1, dy1,`
`dx2, dy2,`
`dx3, dy3)`
	appends a Bézier curve with the 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 only the control box, *i.e.* not neccesarily the smallest enclosing rectangle.

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 series of `pathel`s. A simple example, which 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. This range from standard operation like the determination of the length of a path via `len(p)`, fetching of items using `p[index]` and the possibility to concatenate two paths, `p1 + p2`, append further `pathel`s using `p.append(pathel)` to more advanced methods, which are summarized in the following table.
XXX terminology: subpath, . . .

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

Some notes on the above:

- The bounding box may be too large, if the path contains any `curveto` elements, since for these the control box, *i.e.*, the bounding box enclosing the control points of the Bézier curve is returned.
- The † denotes methods which require a prior conversion of the path into a `normpath` instance. This is done automatically, but if you need many 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 `<<` opertor, 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, namely precisely those which are denoted by a † in the above table.

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

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

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

3.4. Subclasses of path

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

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

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

4. Module trafo: linear transformations

With the `trafo` modulo PyX provides 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 give special operations like translation, rotation, scaling, and mirroring.

4.1. Class trafo

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

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

where A is the transformation matrix and $\vec{b}$ the translation vector. The transformation matrix must not be singular, *i.e.* we require $\det \mathsf{A} \neq 0$.
Multiple `trafo` instances can be multiplied, corresponding to a consecutive application of the respective transformation. Note that `trafo1*trafo2` means that first `trafo2` gets applied and then `trafo1`, *i.e.* the new transformation is given in obvious notation by $\mathsf{A} = \mathsf{A}_1\mathsf{A}_2$ and $\vec{b} = \mathsf{A}_1\vec{b}_2 + \vec{b}_1$. Use the `trafo` methods described below, if you prefer thinking the other way round. The inverse of a transformation can be obtained via the `trafo` method `inverse()`, defined by the inverse A^{-1} of the transformation matrix and the transformation vector $-\mathsf{A}^{-1}\vec{b}$.
The methods of the `trafo` class are summarized in the following table.

trafo method	function

`__init__(matrix=((1,0),(0,1)),`
`vector=(0,0)):`
	create new `trafo` instance with transformation `matrix` and `vector`.

`apply(x, y)`	apply `trafo` to point vector $(\mathtt{x}, \mathtt{y})$.
`inverse()`	returns inverse transformation of `trafo`.
`mirrored(angle)`	returns `trafo` followed by mirroring at line through $(0, 0)$ with direction `angle` in degrees.

`rotated(angle,`
`x=None, y=None)`
	returns `trafo` followed by rotation by `angle` degrees around point $(\mathtt{x}, \mathtt{y})$, or $(0, 0)$, if not given.

`scaled(sx, sy=None,`
`x=None, y=None)`
	returns `trafo` followed by scaling with scaling factor `sx` in x-direction, `sy` in y-direction ($\mathtt{sy} = \mathtt{sx}$, if not given) with scaling center $(\mathtt{x}, \mathtt{y})$, or $(0, 0)$, if not given.

`translated(x, y)`	returns `trafo` followed by translation by vector $(\mathtt{x}, \mathtt{y})$.
`slanted(a, angle=0, x=None, y=None)`	returns `trafo` followed by XXX

4.2. Subclasses of trafo

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

trafo subclass	function
`mirror(angle)`	mirroring at line through $(0, 0)$ with direction `angle` in degrees.

`rotate(angle,`
`x=None, y=None)`
	rotation by `angle` degrees around point $(\mathtt{x}, \mathtt{y})$, or $(0, 0)$, if not given.

`scale(sx, sy=None,`
`x=None, y=None)`
	scaling with scaling factor `sx` in x-direction, `sy` in y-direction ($\mathtt{sy} = \mathtt{sx}$, if not given) with scaling center $(\mathtt{x}, \mathtt{y})$, or $(0, 0)$, if not given.

`translate(x, y)`	translation by vector $(\mathtt{x}, \mathtt{y})$.
`slant(a, angle=0, x=None, y=None)`	XXX

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, the purpose of which is the collection of 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 proceeds then via the construction of a `path`, which can subsequently be drawn on the canvas using the method `stoke()`:

```
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, as well:

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

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

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

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

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

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

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

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

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

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

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

To conclude the section on the drawing of paths, we consider a pretty sophisicated 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.
After you have finished the composition of the canvas, you can write it to a file using the method `writetofile()`. It expects the obligatory argument `filename`, the name of the output file. To write your results to the file ”test.eps” just call it as follows:

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

5.1.2. Methods of the class canvas

The `canvas` class provides the following methods:

`canvas` method	function
`__init__(*args)`	Construct new canvas. `args` can be instances of `trafo.trafo`, `canvas.clip` and/or `canvas.PathStyle`.
`bbox()`	Returns the bounding box enclosing all elements of the canvas.
`draw(path, *styles)`	Generic drawing routine for given `path` on the canvas (*i.e.* `insert`s it together with the necessary `newpath` command, applying the given `styles`. Styles can either be instances of `base.PathStyle` or `canvas.PathDeco` (or subclasses thereof).
`fill(path, *styles)`	Fills the given `path` on the canvas, *i.e.* `insert`s it together with the necessary `newpath`, `fill` sequence, applying the given `styles`. Styles can either be instances of `base.PathStyle` or `canvas.PathDeco` (or subclasses therof).
`insert(*PSOps)`	Inserts one ore more instances of the class `base.PSOp` in the canvas and returns the last one. Thereby, instances of `canvas.canvas` are bracketed by `gsave`/`grestore` pair.
`set(*styles)`	Sets the given `styles` (instances of `base.PathStyle` or subclasses) for the rest of the canvas.
`stroke(path, *styles)`	Strokes the given `path` on the canvas, *i.e.* `insert`s it together with the necessary `newpath`, `stroke` sequence, applying the given `styles`. Styles can either be instances of `base.PathStyle` or `canvas.PathDeco` (or subclasses thereof).

`writetofile(filename,`
`paperformat=None,`
`rotated=0,`
`fittosize=0,`
`margins="1 t cm")`
	Writes the canvas to `filename`. Optionally a `paperformat` can be specified, in which case the output will be centered with respect to the corresponding size using the given `margin`. See `canvas._paperformats` for a list of known paper formats . Use `rotated`, if you want to center on a 90° rotated version of the respective paper format. If `fittosize` is set, the output is additionally scaled to the maximal possible size.

5.2. 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 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("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
`filename`	Name of the EPS file (including a possible extension).
`x="0 t m"`	x-coordinate of position (converts to user unit by default).
`y="0 t m"`	y-coordinate of position (converts to user unit by default).
`width=None`	Desired width of EPS graphics or `None` for original width. Cannot be combined with scale specification.
`heigth=None`	Desired height of EPS graphics or `None` for original height. Cannot be combined with scale specification.
`scale=None`	Scaling factor for EPS graphics or `None` for no scaling. Cannot be combined with width or height specification.
`align="bl"`	Alignment of EPS graphics. The first character specifies the vertical alignment: `b` for bottom, `c` for center, and `t` for top. The second character fixes the horizontal alignment: `l` for left, `c` for center `r` for right.
`clip=1`	Clip to bounding box of EPS file?
`showbbox=0`	Stroke bounding box of EPS file?
`translatebox=1`	Use lower left corner of bounding box of EPS file? Set to 0 with care.

7. Module tex: TEX/LATEX interface

7.1. Methods

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

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

There are some common rules:

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

- All text is inserted into the `canvas` at the position, where the `tex`- or `latex`-instance itself is inserted into the `canvas`. In fact, the `eps`-file created by TEX or LATEX and `dvips` is just inserted.

- The tailing `*style` parameter stands for a list of attribute parameters listed in the last column of the table. Attribute parameters are instances of classes discussed in detail in the following section.

- There can be several `msghandler` attributes which will be applied sequentially. All other parameters can occure only once.

7.2. Attributes

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

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

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

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

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

`color:` stands for any PyX color (see section 8), default is `color.gray.black`

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

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

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

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

7.3. Constructors

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

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

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

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

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

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

7.4. Examples

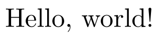
7.4.1. Example 1

```
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:



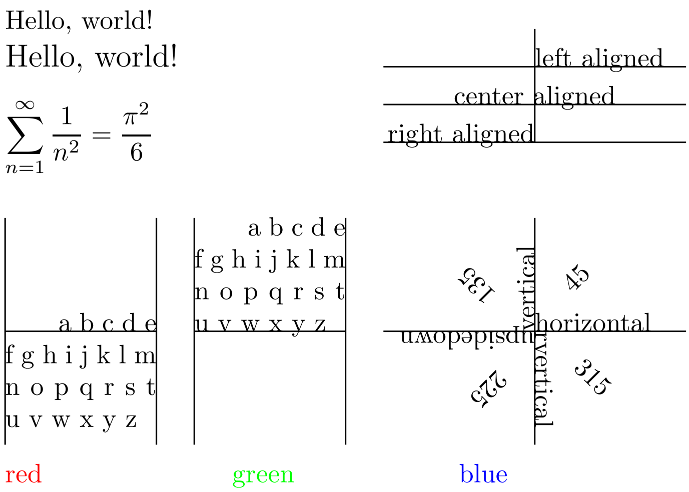
Hello, world!

7.4.2. Example 2

```
from pyx import *

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

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

```
c.stroke(path.line(5, -1, 9, -1))
c.stroke(path.line(5, -1.5, 9, -1.5))
c.stroke(path.line(7, -1.5, 7, 0))

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

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

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

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

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

c.stroke(path.line(5, -4, 9, -4))
c.stroke(path.line(7, -5.5, 7, -2.5))

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

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

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

c.writetofile("tex2")
```

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



7.5. Known bugs

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

- Due to `dvips` the bounding box is wrong for rotated text. The rotation is just ignored in the bounding box calculation.

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

7.6. 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.

8. Module color

8.1. Color models

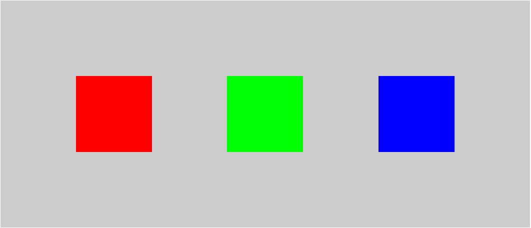
PostScript provides different color models. They are available to PyX by different color classes, which just pass the colors down to the PostScript level. This implies, that there are no conversion routines between different color models available. However, some color model conversion routines are included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

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

8.2. 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:



8.3. Color gradients

The color module provides a class `gradient`. 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 gradients is available in appendix C.

9. Module graph: graph plotting

9.1. Introductory notes

The graph module is considered to be in constant, gradual development. For the moment we concentrate ourself on standard 2d xy-graphs taking all kind of possible specialties into account like any number of axes. Architectural decisions play the most substantial role at the moment and have hopefully already been done that way, that their flexibility will suffice for future usage in quite different graph applications, *e.g.* circular 2d graphs or even 3d graphs. We will describe those parts of the graph module here, which are in a totally usable state already and are hopefully not to be changed later on. However, future developments certainly will cause incompatibilities, for example they are expected to happen for automatic axis ticking (which will therefore not yet be covered within this manual). At least be warned: Nobody knows the hole list of things that will break. At the moment, keeping backwards compatibility in the graph module is not at all an issue. Although we do not yet claim any backwards compatibility for the future at all, the graph module is certainly one of the biggest construction sites within PyX.

The task of drawing graphs is splitted in quite some subtasks, which are implemented by classes of its own. We tried to make those components as independend as it is usefull and possible in order to make them reuseable for different graph types. They are also replaceable by the user to get more specialized graph drawing tasks done without needing to implement a hole graph system. A major abstraction layer are the so-called graph coordinates. Their range is generally fixed to $[0;1]$. Only the graph does know about the conversion between these coordinates and the position at the canvas. By that, all other components can be reused for different graph geometries.

9.2. Axes

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

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

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

9.2.1. Axes properties

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

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

9.2.2. Partitioning of axes

The definition of ticks and labels appropriate to an axis range is called partitioning. The axis partioning within PyX uses rational arithmetics, which avoids any kind of rounding problems to the cost of performance. The class `frac` supplies a rational number. 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 partioning schemes.

Manual partitioning

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

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

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 descibes
`shift5fracs1`	1 and multiple of 10^5
`shift4fracs1`	1 and multiple of 10^4
`shift3fracs1`	1 and multiple of 10^3
`shift2fracs1`	1 and multiple of 10^2
`shiftfracs1`	1 and multiple of 10
`shiftfracs125`	1, 2, 5 and multiple of 10
`shiftfracs1to9`	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
`list`	`defaultlist`	list of possible values for the ticks parameter of `linpart` (labels are placed at the position of ticks)
`extendtick`	`0`	allow for a range extention to include the next tick of the given level
`epsilon`	`1e-10`	allow for exceeding the range by that relative value
`mix`	`()`	as in manualpart

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

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

```
(((shiftfracs1, shiftfracs1to9),        # ticks & subticks,
         (shiftfracs1, shiftfracs125)), # labels & sublevels
 ((shiftfracs1, shiftfracs1to9), None), # ticks & subticks, labels=ticks
 ((shift2fracs1, shiftfracs1), None),   # ticks & subticks, labels=ticks
 ((shift3fracs1, shiftfracs1), None),   # ticks & subticks, labels=ticks
 ((shift4fracs1, shiftfracs1), None),   # ticks & subticks, labels=ticks
 ((shift5fracs1, shiftfracs1), 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 hole system goes beyond the scope of that manual. Take your freedom and have a look at the PyX source code if you wish to adopt the rating to personal preferences.

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

9.2.3. 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
`innerticklengths`[1,4]	tick length of inner ticks (visual length); default: `axispainter.defaultticklengths`
`outerticklengths`[1,4]	as before, but for outer ticks; default: `None`
`tickattrs`[2,4]	stroke attributes for ticks; default: `()`
`gridattrs`[2,4]	stroke attributes for grid lines; default: `None`
`zerolineattrs`[3,4]	stroke attributes for a grid line at axis value 0; default: `()`
`baselineattrs`[3,4]	stroke attributes for the axis baseline; default: `canvas.linecap.square`
`labeldist`	label distance from axis (visual length); default: `"0.3 cm"`
`labelattrs`[2,4]	text attributes for labels; default: `((), tex.fontsize.footnotesize)`
`labeldirection`[4]	relative label direction (see below); default: `None`
`labelhequalize`	set width of labels to its maximum (boolean); default: `0`
`labelvequalize`	set height and depth of labels to their maxima (boolean); default: `1`
`titledist`	title distance from labels (visual length); default: `"0.3 cm"`
`titleattrs`[3,4]	text attributes for title; default: `()`
`titledirection`[4]	relative title direction (see below); default: `axispainter.paralleltext`
`titlepos`	title position in graph coordinates; default: `0.5`
`fractype`	text creation for labels (see below); default: `axispainter.fractypeauto`
`ratfracsuffixenum`	write suffix at the enumerator (boolean); default: `1`
`ratfracover`	text for fraction line; default: `r"\over"`
`decfracpoint`	decimal point; default: `"."`
`expfractimes`	text between factor and decimal power; default: `r"\cdot"`
`expfracpre1`	allow factor 1 before a decimal power (boolean); default: `0`
`expfracminexp`	minimal exponent for decimal power; default: `4`
`suffix0`	when a suffix is `x` write `0x` instead of `0` (boolean); default: `0`
`suffix1`	when a suffix is `x` write `1x` instead of `x` (boolean); default: `0`

[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.paralleltext` 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.

`fractype`	description	example
`axispainter.fractypedec`	decimal	0.1
`axispainter.fractypeexp`	decimal with exponent	$2 \cdot 10^4$
`axispainter.fractyperat`	rational	$\frac{1}{2}$
`axispainter.fractypeauto`	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.

9.2.4. 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 `zerolineattrs` 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.

9.2.5. 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)
`splitlist`	a single number or a list split points of the possitions of the axis breaks in graph coordinates; the value `None` forces `relsizesplitdist` to be used; default: `0.5`
`splitdist`	gap of the axis break; default: `0.1`
`relsizesplitdist`	used when `splitlist` entries are `None`; gap of the axis break in values of the surrounding axes (on logarithmic axes, a decade corresponds to 1); the split position is adjusted to give both surrounding axes the same scale (thus, their range must be completely fixed); default: `1`
`title`	axis title
`painter`	axis painter; default: `splitaxispainter()` (described below)

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

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

Additionally, the painter takes parameters for the axis title formatting like the standard axis painter class `axispainter`. The parameters are `titledist`, `titleattrs`, `titledirection`, and `titlepos`.

Bar axes

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

argument name	description
subaxis	baraxis can be recursive by having another axis as its subaxis; default: None
multisubaxis	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: 0.5*dist
lastdist	as before but for the last bar
names	tuple of name identifiers for bars; when set, no other identifiers are allowed; default: None
texts	dictionary translating names into label texts (otherwise just the names are used); default: {}
painter	axis painter; default: baraxispainter (described below)

In contrast to other axes, a bar axis uses name identifiers to calculate a possition at the axis. Usually, a style appropriate to a bar axis (this is right now just the bar style) set those names out of the data it recieves. However, the names can be forced and fixed.
Bar axes can be recursive. Thus for a given value, an appropriate subaxis is choosen (usually another bar axis). Usually only a single subaxis is needed, because 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 as the axispainter. Thus the constructor have quite some parameters in common, namely titledist, titleattrs, titledirection, titlepos, and baselineattrs. Furthermore the two 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 hole bunch of name attribute identifiers, namely namedist, nameattrs, namedirection, namehequalize, namevequalize which are identical to their counterparts called label... instead of name. Last but not least, there is a parameter namepos which is analogous to titlepos and set to 0.5 by default.

9.3. Data

9.3.1. List of points

Instances of the class data link a datafile and a style (see below; default is symbol). The link object is needed in order to be able to plot several data from a singe file without reading the file several times which would just be a bad design. However, for easy usage, it is possible to provide a filename instead of a datafile as the first argument to the

constructor of the class `data` hiding the underlying `datafile` instance completely from view. This is the preverable 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 expressions, as the following table will show by some examples.

selection method	example

as in `datafile.getcolumnno`	`data("test.dat", x=1,`
	`y="result", dy="delta")`

by mathematical expressions	`data("test.dat", x="0.5*$1",`
	`y="0.5*result", dy="0.5*a", a=3)`

Note that mathematical expressions get evaluated by `datafile.addcolumn` and thus the same column identifications become available.

9.3.2. 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 `extern` argument.
Additional named parameters of the constructor are:

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

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

9.3.3. 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`, and `extern` behave like their equally named counterparts in `function`.

9.4. 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.

9.4.1. Symbols

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

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

The parameter `symbol` has to be a routine, which returns a path to be drawn (e.g. stoked or filled). There are several those 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 9.4.7 for details.

The attributes are available as class variables after plotting the style for outside usage. Additionally, the variable `path` contains the path of the line (even when it wasn't plotted), which might be used to get crossing points, fill areas, etc.

Valid data names to be used when providing data to symbols are listed in the following table. The character `X` stands for axis names like `x`, `x2`, `y`, etc.

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

9.4.2. Lines

The class `line` is inherited from `symbol` and is restricted to line drawing. The constructor takes only `lineattrs` and its default is set to `changelinestyle()`. The other features of the symbol style are turned off.

9.4.3. 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 gradient provided to the constructor by the named parameter `gradient` (default is `color.gradient.Gray`). The data name `color` is used to select the color out of this gradient.

9.4.4. 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 `tex.halign.center`. Furthermore the constructor of this class allows all other attributes of the class `symbol`.

9.4.5. Arrows

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

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

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

9.4.6. Bars

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

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

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

9.4.7. 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 gradient. Applying this attribute to a style and using this style at a sequence of data, the color will get changed lineary along the gradient from one end to the other. The class `colorchange` includes inherited classes as class variables, which are called like the color gradients shown in appendix C. For them the default gradient is set to the appropriate color gradient.
Another attribute changer is called `changesequence`. The constructor takes a list of attributes and the attribute changer cycles through this list whenever a new attribute is requested. This attribute changer is used to implement the following attribute changers:

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

The class `changesymbol` can be used to cycle throu symbols and it provides already various specialized classes as class variables. To loop over all available symbols (cross, plus, square, triangle, circle, and diamond) the equal named class variables can be used. They start at that symbol they are named of. Thus `changesymbol.cross()` cycles throu the 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 intented to be used in combination with `changestrokedfilled` and `changefilledstroked`.

9.5. Keys

Sorry, there is not yet any support for graph keys.

9.6. 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 9.2. 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, namely first of all a tex canvas. (This ugly construction is likely to be ommited in future versions of PyX once a new TEX binding becomes available.) Other parameters are named and listed in the following table:

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

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 9.3. 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 style (or a list of styles when a data list was provided) which was used for plotting. Just as an example you can thus access the path of a line and fill areas with it and so on.

After the plot method was called once or several times, you should call the method `finish`. (This is actually needed as long as a tex canvas gets used for text output and the tex canvas is inserted into the main canvas before the graph gets inserted.) Finishing a graph allows for the access to positioning routines which can be quite usefull to plot additional information into a graph.

Sometimes it is also nice to partly finish a graph. By that you can even modify the order in which a graph performs its drawing process. By default the four methods `dolayout`, `dobackground`, `doaxis`, and `dodata` 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 named parameters `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 gridpath for a given position at the axis.

9.7. Examples

9.7.1. A minimal example: plot data from a file

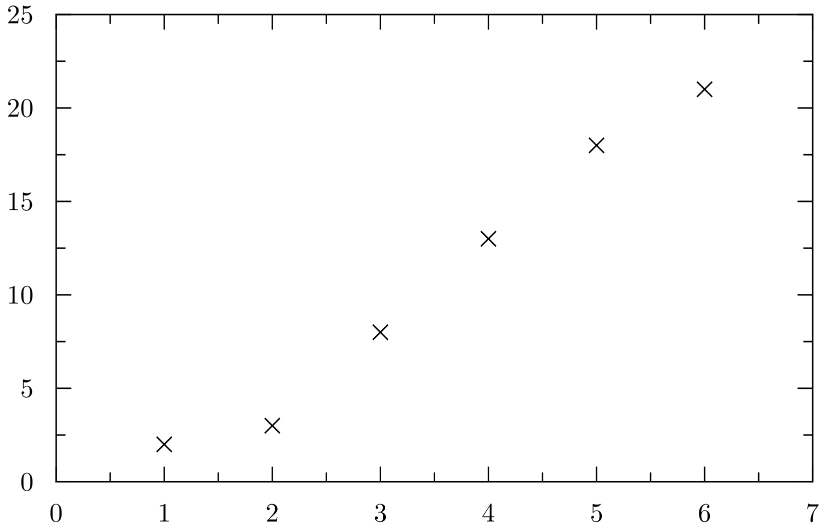
We plot data from the file `"graph.dat"`:

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

The following script creates the file `"graph.eps"`:

```
from pyx import *

c = canvas.canvas()
t = c.insert(tex.tex())
g = c.insert(graph.graphxy(t, width=10))
g.plot(graph.data("graph.dat", x=1, y=2))
g.finish()
c.writetofile("graph")
```

The result looks like:

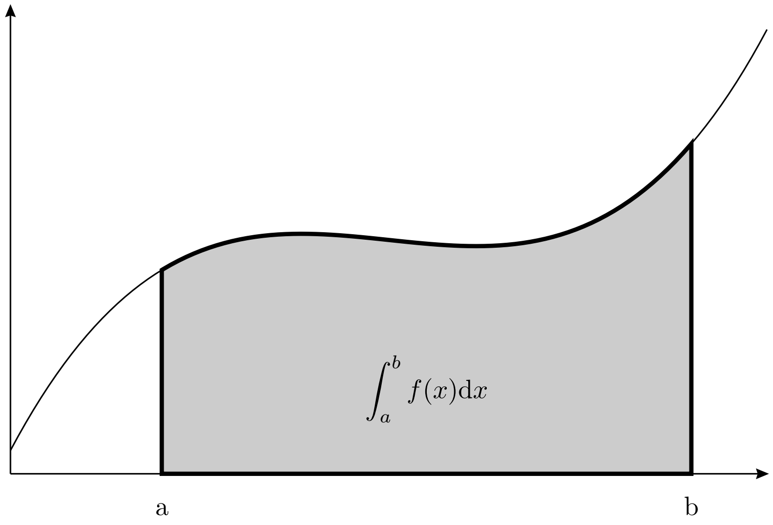


9.7.2. A more advanced function plot

```
from pyx import *
from pyx.graph import *

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

a, b = 2, 9

mypainter=axispainter(baselineattrs=canvas.earrow.normal)

g = c.insert(graphxy(t, width=10, x2=None, y2=None,
                     x=linaxis(min=0, max=10,
                               part=manualpart(ticks=(frac(a, 1),
                                                       frac(b, 1)),
                                                texts=("a", "b")),
                               painter=mypainter),
                     y=linaxis(painter=mypainter,
                               part=manualpart())))

line = g.plot(function("y=(x-3)*(x-5)*(x-7)"))
g.finish()

pa = path.path(g.axes["x"].gridpath(a))
pb = path.path(g.axes["x"].gridpath(b))
(splita,), (splitpa,) = line.path.intersect(pa)
(splitb,), (splitpb,) = line.path.intersect(pb)
```

```
area = (pa.split(splitpa)[0] <<
        line.path.split(splita, splitb)[1] <<
        pb.split(splitpb)[0].reversed())
area.append(path.closepath())
g.stroke(area, canvas.linewidth.THick,
         canvas.filled(color.gray(0.8)))
t.text(g.pos(0.5*(a+b), 0)[0], 1,
       r"\int_a^b f(x) {\rm d}x", tex.halign.center, tex.style.math)

c.insert(t)
c.writetofile("graph2")
```

The result looks like:



A. Mathematical expressions

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

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

Functions: `neg` (negate); `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
grey.white

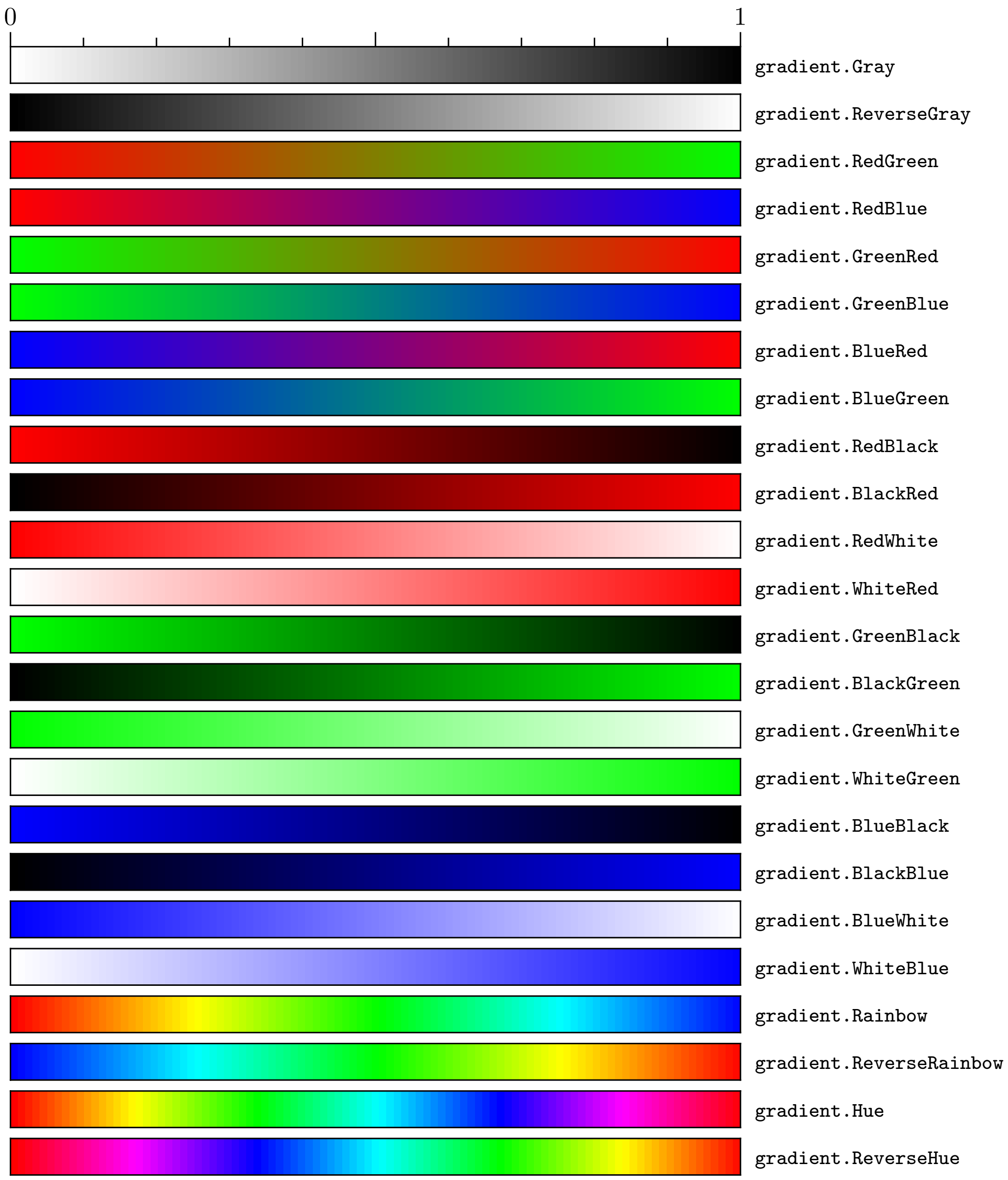
rgb.red
rgb.green
rgb.blue

cmyk.GreenYellow
cmyk.Yellow
cmyk.Goldenrod
cmyk.Dandelion
cmyk.Apricot
cmyk.Peach
cmyk.Melon
cmyk.YellowOrange
cmyk.Orange
cmyk.BurntOrange
cmyk.Bittersweet
cmyk.RedOrange
cmyk.Mahogany
cmyk.Maroon
cmyk.BrickRed
cmyk.Red
cmyk.OrangeRed
cmyk.RubineRed

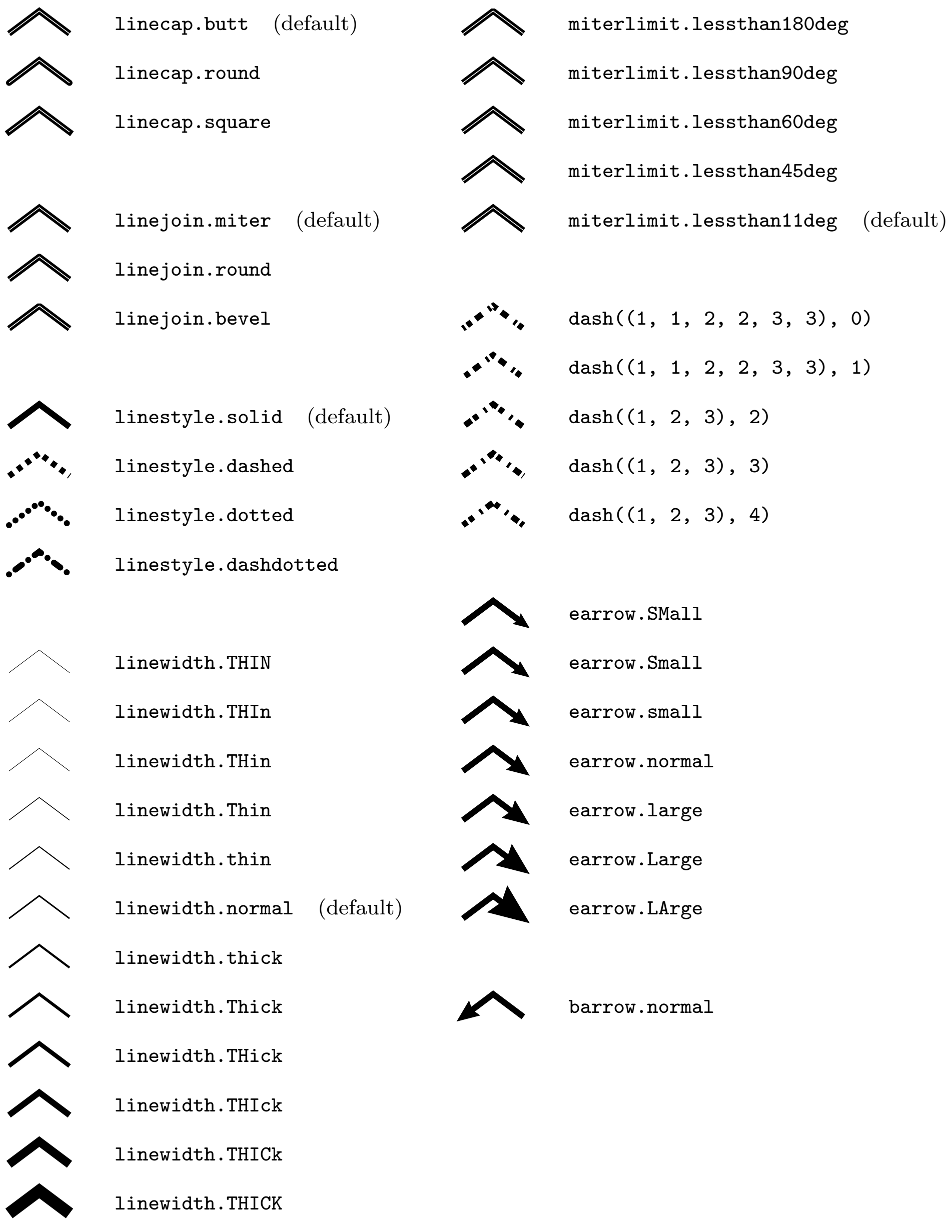
cmyk.WildStrawberry
cmyk.Salmon
cmyk.CarnationPink
cmyk.Magenta
cmyk.VioletRed
cmyk.Rhodamine
cmyk.Mulberry
cmyk.RedViolet
cmyk.Fuchsia
cmyk.Lavender
cmyk.Thistle
cmyk.Orchid
cmyk.DarkOrchid
cmyk.Purple
cmyk.Plum
cmyk.Violet
cmyk.RoyalPurple
cmyk.BlueViolet
cmyk.Periwinkle
cmyk.CadetBlue
cmyk.CornflowerBlue
cmyk.MidnightBlue
cmyk.NavyBlue
cmyk.RoyalBlue
cmyk.Blue

cmyk.Cerulean
cmyk.Cyan
cmyk.ProcessBlue
cmyk.SkyBlue
cmyk.Turquoise
cmyk.TealBlue
cmyk.Aquamarine
cmyk.BlueGreen
cmyk.Emerald
cmyk.JungleGreen
cmyk.SeaGreen
cmyk.Green
cmyk.ForestGreen
cmyk.PineGreen
cmyk.LimeGreen
cmyk.YellowGreen
cmyk.SpringGreen
cmyk.OliveGreen
cmyk.RawSienna
cmyk.Sepia
cmyk.Brown
cmyk.Tan
cmyk.Gray
cmyk.Black
cmyk.White

C. Named gradients



D. Path styles and arrows in canvas module


linecap.butt (default)
linecap.round
linecap.square
linejoin.miter (default)
linejoin.round
linejoin.bevel
linestyle.solid (default)
linestyle.dashed
linestyle.dotted
linestyle.dashdotted
linewidth.THIN
linewidth.THIn
linewidth.THin
linewidth.Thin
linewidth.thin
linewidth.normal (default)
linewidth.thick
linewidth.Thick
linewidth.THick
linewidth.THIck
linewidth.THICk
linewidth.THICK
miterlimit.lessthan180deg
miterlimit.lessthan90deg
miterlimit.lessthan60deg
miterlimit.lessthan45deg
miterlimit.lessthan11deg (default)
dash((1, 1, 2, 2, 3, 3), 0)
dash((1, 1, 2, 2, 3, 3), 1)
dash((1, 2, 3), 2)
dash((1, 2, 3), 3)
dash((1, 2, 3), 4)
earrow.SMall
earrow.Small
earrow.small
earrow.normal
earrow.large
earrow.Large
earrow.LArge
barrow.normal