

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

PyX 0.1

User Manual

Jörg Lehmann <joergl@users.sourceforge.net>

André Wobst <wobsta@users.sourceforge.net>

October 7, 2002

PostScript is a trademark of Adobe Systems Incorporated.

Contents

1. Introduction	5
2. Module unit	6
2.1. Class length	6
2.2. Subclasses of length	7
2.3. Conversion functions	7
3. Module path: PostScript like paths	8
3.1. Class pathel	8
3.2. Class path	9
3.3. Class normpath	10
3.4. Subclasses of path	11
4. Module trafo: linear transformations	12
4.1. Class trafo	12
4.2. Subclasses of trafo	13
5. Module canvas: PostScript interface	14
5.1. Class canvas	14
5.1.1. Basic usage	14
5.1.2. Methods of the class canvas	15
5.2. Subclasses of base.PathStyle	17
6. Module epsfile: EPS file inclusion	18
7. Module tex: T_EX/L^AT_EX interface	19
7.1. Methods	19
7.2. Attributes	20
7.3. Constructors	21
7.4. Examples	22
7.4.1. Example 1	22
7.4.2. Example 2	22
7.5. Known bugs	24
7.6. Future of the module tex	24

8. Module color	25
8.1. Color models	25
8.2. Example	25
8.3. Color gradients	26
9. Module datafile: reading a datafile	27
9.1. Reading a table from a file	27
9.2. Accessing columns	28
9.3. Mathematics on columns	28
9.4. Dirty tricks for mathematics on columns	29
9.5. Own datafile readers	29
10. Module graph: graph plotting	31
10.1. Introductory notes	31
10.2. Axes	31
10.2.1. Axes properties	32
10.2.2. Partitioning of axes	32
10.2.3. Painting of axes	34
10.2.4. Linked axes	36
10.3. Data	36
10.3.1. List of points	36
10.3.2. Functions	37
10.3.3. Parametric functions	37
10.4. Styles	38
10.4.1. Marks	38
10.4.2. Lines	39
10.4.3. Rectangles	39
10.4.4. Texts	39
10.4.5. Arrows	39
10.4.6. Iterateable style attributes	40
10.5. Keys	40
10.6. X-Y-Graph	41
10.7. Examples	42
10.7.1. A minimal example: plot data from a file	42
10.7.2. A more advanced function plot	43
A. Mathematical expressions	45
B. Named colors	46
C. Named gradients	47
D. Path styles and arrows in canvas module	48

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 $\text{T}_{\text{E}}\text{X}/\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ interface enables one to use the famous high quality typesetting within the figures.

A major part of $\text{P}_{\text{Y}}\text{X}$ 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 <code>length</code>	Type	Unit	Subclass of <code>length</code>	Type	Unit
<code>m(x)</code>	user	m	<code>t_m(x)</code>	true	m
<code>cm(x)</code>	user	cm	<code>t_cm(x)</code>	true	cm
<code>mm(x)</code>	user	mm	<code>t_mm(x)</code>	true	mm
<code>inch(x)</code>	user	inch	<code>t_inch(x)</code>	true	inch
<code>pt(x)</code>	user	points	<code>t_pt(x)</code>	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
<code>to_m(l)</code>	1 in units of m
<code>to_cm(l)</code>	1 in units of cm
<code>to_mm(l)</code>	1 in units of mm
<code>to_inch(l)</code>	1 in units of inch
<code>to_pt(l)</code>	1 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 `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 relative by (dx, dy)
<code>lineto(x, y)</code>	appends straight line from current point to (x, y)
<code>rlneto(dx, dy)</code>	appends straight line from current point relative by (dx, dy)
<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 with radius r which connects between (x1, y1) and (x2, y2).
<code>rcurveto(dx1, dy1, dx2, dy2, dx3, dy3)</code>	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 necessarily 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 `pathels`. 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 `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 path consisting of pathels
<code>append(pathel)</code>	appends pathel to end of path
<code>arclength(epsilon=1e-5)</code>	returns the total arc length of all path segments in PostScript points with accuracy epsilon . [†]
<code>at(t)</code>	returns the coordinates of the point of path corresponding to the parameter value t . [†]
<code>bbox()</code>	returns the bounding box of the path
<code>begin()</code>	return first point of first subpath of path . [†]
<code>end()</code>	return last point of last subpath of path . [†]
<code>glue(opath)</code>	returns the path glued together with opath , <i>i.e.</i> the last subpath of path and the first one of opath are joined. [†]
<code>intersect(opath, epsilon=1e-5)</code>	returns tuple consisting of two list of parameter values corresponding to the intersection points of path and opath , respectively. [†]
<code>reversed()</code>	returns the normalized reversed path . [†]
<code>split(t)</code>	returns a tuple consisting of two normpaths corresponding to the path split at the parameter value t . [†]
<code>transformed(trafo)</code>	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 `<<` 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, 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 `pathels` 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 <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` 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

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

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 first `trafo2` gets applied and then `trafo1`, *i.e.* the new transformation is given in obvious notation 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 transformation 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 <code>trafo</code> instance with transformation matrix and vector.
<code>apply(x, y)</code>	apply <code>trafo</code> to point vector (x,y).
<code>inverse()</code>	returns inverse transformation of <code>trafo</code> .
<code>mirror(angle)</code>	returns <code>trafo</code> followed by mirroring at line through (0,0) with direction <code>angle</code> in degrees.
<code>rotate(angle, x=None, y=None)</code>	returns <code>trafo</code> followed by rotation by <code>angle</code> degrees around point (x,y), or (0,0), if not given.
<code>scale(sx, sy=None, x=None, y=None)</code>	returns <code>trafo</code> followed by scaling with scaling factor <code>sx</code> in <i>x</i> -direction, <code>sy</code> in <i>y</i> -direction (<code>sy = sx</code> , if not given) with scaling center (x,y), or (0,0), if not given.
<code>translate(x, y)</code>	returns <code>trafo</code> followed by translation by vector (x,y).

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:

<code>trafo</code> subclass	function
<code>mirroring(angle)</code>	mirroring at line through (0,0) with direction <code>angle</code> in degrees.
<code>rotation(angle, x=None, y=None)</code>	rotation by <code>angle</code> degrees around point (x,y), or (0,0), if not given.
<code>scaling(sx, sy=None, x=None, y=None)</code>	scaling with scaling factor <code>sx</code> in <i>x</i> -direction, <code>sy</code> in <i>y</i> -direction (<code>sy = sx</code> , if not given) with scaling center (x,y), or (0,0), if not given.
<code>translation(x, y)</code>	translation by vector (x,y).

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, T_EX or L^AT_EX 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 `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, 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 `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.

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
<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(*PSOps)</code>	Inserts one or more instances of the class <code>base.PSOp</code> in the canvas and returns the last one. Thereby, instances of <code>canvas.canvas</code> are bracketed by <code>gsave/grestore</code> pair.
<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>writetofile(filename, paperformat=None, rotated=0, fittosize=0, margins="1 t cm")</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.

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
<code>filename</code>	Name of the EPS file (including a possible extension).
<code>x="0 t m"</code>	<i>x</i> -coordinate of position (converts to user unit by default).
<code>y="0 t m"</code>	<i>y</i> -coordinate of position (converts to user unit by default).
<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.

7. Module `tex`: $\text{T}_{\text{E}}\text{X}$ / $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ interface

7.1. Methods

Text in $\text{P}_{\text{Y}}\text{X}$ is created by $\text{T}_{\text{E}}\text{X}$ or $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$. 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 $\text{T}_{\text{E}}\text{X}$ or $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ followed by `dvips` to create the requested text. $\text{T}_{\text{E}}\text{X}$ is used by instances of the class `tex` while $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ is used by `latex`. Up to the constructor and the advanced possibilities in $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ 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	style, fontsize, halign, valign, direction, color, msghandler(s)
<code>define(cmd, *attr)</code>	execute cmd	msghandler(s)
<code>textwd(cmd, *attr)</code>	width of cmd	style, fontsize, missextents, msghandler(s)
<code>textht(cmd, *attr)</code>	height of cmd	style, fontsize, valign, missextents, msghandler(s)
<code>textdp(cmd, *attr)</code>	depth of cmd	style, fontsize, valign, missextents, msghandler(s)

There are some common rules:

- `cmd` stands for a $\text{T}_{\text{E}}\text{X}$ or $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ 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 $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ 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 $\text{P}_{\text{Y}}\text{X}$ 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 $\text{T}_{\text{E}}\text{X}$ 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 `TEX` or `LATEX` 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.

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`, `upsideown`, 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`

`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>T_EX</code> or <code>L^AT_EX</code> immediately to get the requested size
<code>misextents.createallextent</code>	run <code>T_EX</code> or <code>L^AT_EX</code> immediately to get the hight, width, and depth of the given text at once

`msghandler`: provides a filter for $\text{T}_{\text{E}}\text{X}$ and $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ 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 $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ 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

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
<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 $\text{T}_{\text{E}}\text{X}$ or $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$. 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
docclass	"article"	specifies the document class
docopt	None	specifies options to the document class
auxfilename	None	Specifies a filename for storing the L ^A T _E X 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, "upside down", tex.direction.upsidedown)

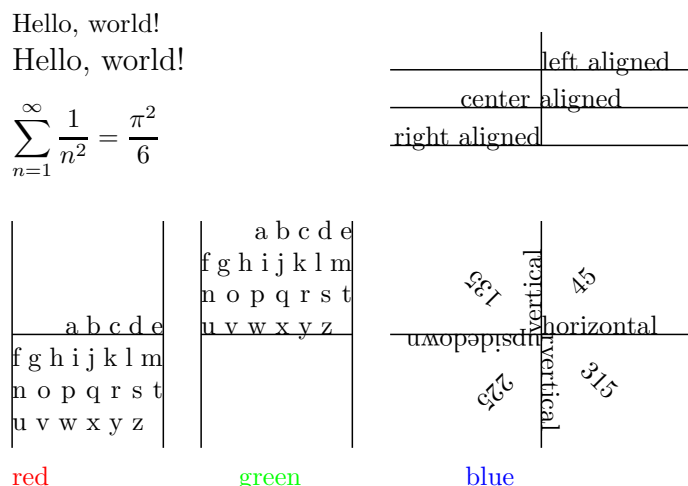
t.text(7.5, -3.5, "45", tex.direction(45))
t.text(6.5, -3.5, "135", tex.direction(135))
t.text(6.5, -4.5, "225", tex.direction(225))
t.text(7.5, -4.5, "315", tex.direction(315))

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 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 splitted 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 occur 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 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.

8.2. Example

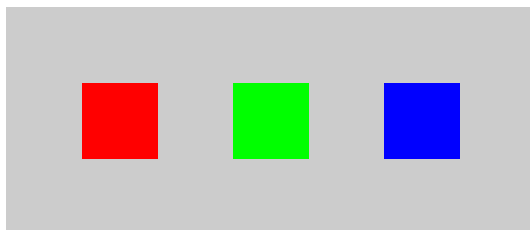
```
from pyx import *

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 datafile: reading a datafile

9.1. 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, namely:

argument name	default	description
<code>commentpattern</code>	<code>re.compile(r"(\#+ !+ %+)\s*")</code>	start a comment line
<code>stringpattern</code>	<code>re.compile(r"\"(.*)\"(\s+ \$)")</code>	a string column
<code>columnpattern</code>	<code>re.compile(r"(.*)\"(\s+ \$)")</code>	any other column

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 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 the string pattern and replace the quote character.

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.

9.2. 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.

9.3. 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 and operators can be found in appendix A. 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 and contain only letters, digits and the underscore).
- A temporary name might be used, which is given by additional named parameters of the `addcolumn` method. The named parameters are stronger compared to column titles of the previous point in the determination of the appropriate column. The name of a named parameter will be the variable name in the expression. The value of a named parameter should be a valid parameter of the `getcolumnno` method.
- A variable name can start with the dollar symbol (`$`) and the following integer number will directly refer to a column number. To implement this third possibility, the mathematical expression parser is extended within the datafile module (cf. section 9.4).

The data referenced by variables in the expression need to be floats, otherwise the result for that data line will be `None`. Examples are `addcolumn("av=(min+max)/2")`, `addcolumn("av=(a+b+$3)/2", a=1, b="max")`.

9.4. Dirty tricks for mathematics on columns

I want to present a solution for cumulating data in a new column. As told in the title of this section, it is considered to be a dirty trick, because it relies on side effects in the calculation of the new column, namely it sums up the result in a hidden variable. While that is wanted, it is nothing I would ever consider to do officially (don't expect the following lines to become part of PyX in the future). On the other hand, nothing could be told about using this trick and I don't expect that this feature will break in future versions. Somehow, it is needed and the possibility to implement this trick has to stay.

What we want to do is to add another function within the allowed expression syntax for doing mathematics on columns. We supply a class which we can hook it into the mathematical expression parser later on:

```
from pyx import mathtree

class Cumulate(mathtree.MathTreeFunc1):

    def __init__(self, *args):
        mathtree.MathTreeFunc1.__init__(self, "cumulate", *args)
        self.sum = 0

    def Calc(self, VarDict):
        self.sum += self.ArgV[0].Calc(VarDict)
        return self.sum

MyFuncs = mathtree.DefaultMathTreeFuncs + (Cumulate,)
```

Please note, that you explicitly have to import `mathtree`, because its not usually needed in PyX applications and thus it is not imported by `from pyx import *`.

To finally use `cumulate`, you have to supply a new parser to the datafile, where the new generated list of available functions is placed in:

```
df = datafile.datafile("mydata",
    parser=mathtree.parser(
        MathTreeFuncs=MyFuncs,
        MathTreeVals=datafile.MathTreeValsWithCol))
df.addcolumn("sum=cumulate(costs)")
```

The explicit setting of `MathTreeVals` is needed in order to keep column variables working. Variables starting with the dollar symbol (\$) are not allowed within the original `mathtree`.

9.5. Own datafile readers

The development of other datafile readers should be based on the helper class `_datafile` 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 the title list and a list of data points. A data point itself is a list of floats or strings. The number of entries per data point and the number of titles provided must fit together.

10. Module graph: graph plotting

10.1. 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 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 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.

10.2. 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

Additional axis types are likely to be added in the future.

10.2.1. Axes properties

Global properties of an axis are set as named parameters in the axis constructor. Both predefined axis, 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 (its default value is 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 is 0.05
<code>datavmax</code>	as above, but for the maximum; default is 0.95
<code>tickvmin</code>	minimal graph coordinate for placing ticks to the axis; default is 0
<code>tickvmax</code>	as above, but for the maximum; default is 1
<code>part</code>	axis partitioning (described below)
<code>painter</code>	axis painter (described below)

10.2.2. 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`, `labellevel`. If 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.

Although there is a rudimentary automatic axis partitioning, the recommended solution at the moment is a manual axis partitioning, because the manual axis partitioning will hopefully not break in future versions, while the automatic axis breaking will change for sure at least in the results it creates.

There are three different manual partition schemes, a manual partition, another appropriate for linear axes and a third one for logarithmic axes.

Manual partitioning

The class `manualpart` creates a manual partition 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, then 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.

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

When no explicit axis partitioning is given, an automatic axis partitioning is already available, but it is still considered to be under development. A major feature is missing, namely the rating of possible partitions does not yet attend the label texts, which is important in order to avoid overlap of label texts. In order to provide it, the rating of axis partitions has to be moved into the axis painter. That has just to be done and is considered for the next major release of `PxX`.

10.2.3. Painting of axes

A major task of an axis is the painting of itself. It is done by instances of `axispainter`, provided to the constructor of an axis as its painter. 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>((), tex.fontsize.footnotesize)</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>itledist</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>itledirection</code> ⁴	relative title direction (see below); default: <code>axispainter.paralleltex</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>expfracimes</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 (`itledirection`) are basi-

cally 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.

<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.

10.2.4. 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.

10.3. Data

10.3.1. List of points

Instances of the class `data` link a `datafile` and a `style` (see below; default is `mark`). The link object is needed in order to be able to plot several data from a single 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 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 expressions, as the following table will show by some examples.

selection method	example
as in <code>datafile.getcolumnno</code>	<code>data("test.dat", x=1, y="result", dy="delta")</code>
by mathematical expressions	<code>data("test.dat", x="0.5*\$1", y="0.5*result", dy="0.5*a", a=3)</code>

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

10.3.2. 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 may use the string format syntax to insert external parameters, *e.g.* `"y=sin(%f*x)" % a` where `a` is a float variable.

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	<code>mathtree.parser()</code>	parser for the mathematical expression

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

10.3.3. 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 two named parameters `points` and `parser` behave like their equally named counterparts in `function`.

10.4. Styles

Styles are used to draw data at a graph. A style determines what is painted and how it is painted. Due to this powerful approach there are already some different marker types available and the possibility to introduce other styles opens even more prospects. On the other hand there is not yet any support for bar graphs. This is due to the fact that it might be better implemented together with some specialized axes. It will be shown in the future, what solution will arise out of that idea instead of an disposable implementation right now.

10.4.1. Marks

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

argument name	default	description
<code>mark</code>	<code>changemark.cross()</code>	marker to be used (see below)
<code>size</code>	<code>"0.2 cm"</code>	size of the marker (visual length)
<code>markattrs</code>	<code>canvas.stroked()</code>	draw attributes for the marker
<code>errorscale</code>	<code>0.5</code>	size of the errorbar caps (relative to the marker size)
<code>errorbarattrs</code>	<code>()</code>	stroke attributes for the errorbars
<code>lineattrs</code>	<code>None</code>	stroke attributes for the line

The parameter `mark` 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 `mark`, namely `cross`, `plus`, `square`, `triangle`, `circle`, and `diamond`. Furthermore, changeable attributes might be used here (like the default value `changemark.cross`), see section 10.4.6 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 markers 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 marker
<code>Xmin</code>	minimum for the errorbar
<code>Xmax</code>	maximum for the errorbar
<code>dX</code>	relative size of the errorbar: $Xmin, Xmax = X-dX, X+dX$
<code>dXmin</code>	relative minimum $Xmin = X-dXmin$
<code>dXmax</code>	relative maximum $Xmax = X+dXmax$

10.4.2. Lines

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

10.4.3. 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 `mark`. Indeed, the class `mark` reuses most of the marker code by inheritance, while modifying the errorbar look into a colored filled rectangle and turning off the marker 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.

10.4.4. Texts

Another style to be used within graphs is the class `text`, which adds the output of text to the class `mark`. The text position relative to the markers 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 `mark`.

10.4.5. 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 mark 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).

10.4.6. 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
<code>changelinestyle</code>	iterates linestyles solid, dashed, dotted, dasheddotted
<code>changestrokedfilled</code>	iterates (<code>canvas.stroked()</code> , <code>canvas.filled()</code>)
<code>change-filled-stroked</code>	iterates (<code>canvas.filled()</code> , <code>canvas.stroked()</code>)

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

10.5. Keys

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

10.6. 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 10.2. 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 omitted 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
<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>	<code>None</code>	width of the graph area (axes are outside of that range)
<code>height</code>	<code>None</code>	as abovem, but for the height
<code>ratio</code>	<code>goldenrule</code>	width/height ratio when only a width or height is provided
<code>backgroundattrs</code>	<code>None</code>	background attributes for the graph area
<code>axisdist</code>	"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 10.3. 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.

10.7. Examples

10.7.1. A minimal example: plot data from a file

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

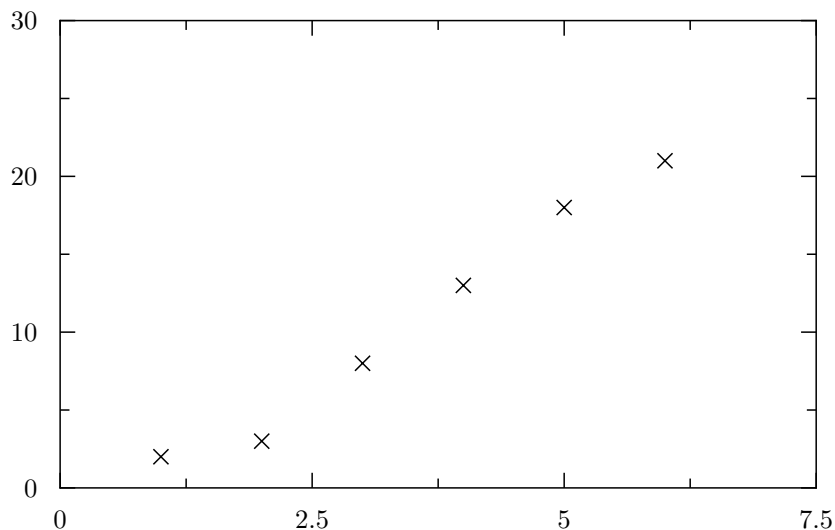
```
1  2
2  3
3  8
4 13
5 18
6 21
```

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:



10.7.2. 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)

```

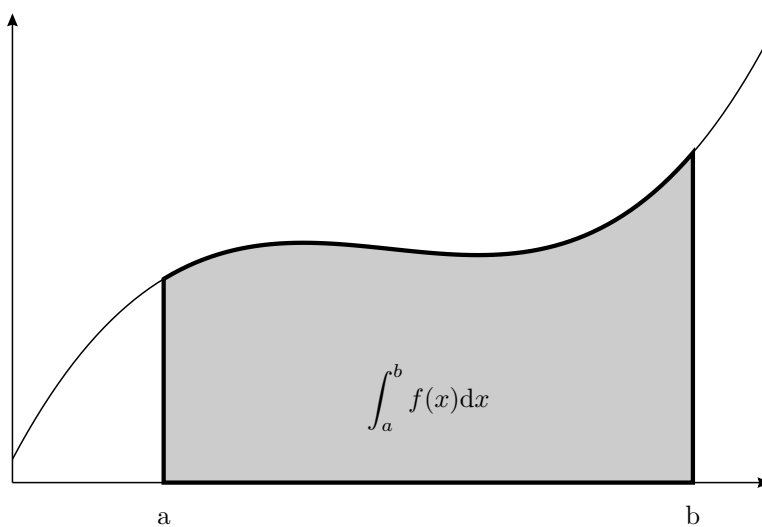
```

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 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)

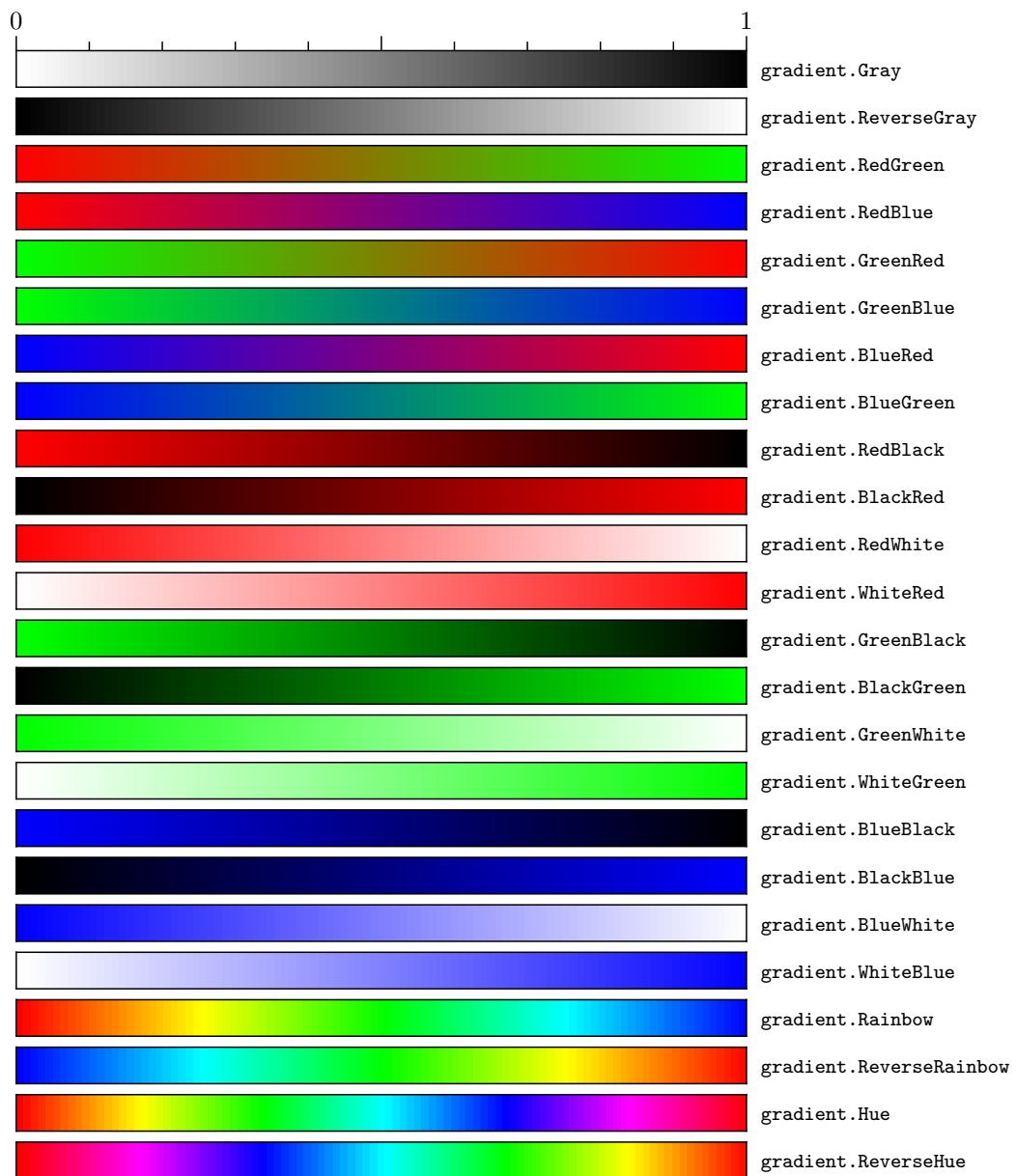
Functions: `neg` (negate); `sgn` (signum); `sqrt` (square root); `exp`; `log` (natural logarithm); `sin`, `cos`, `tan`, `asin`, `acos`, `atan` (trigonometric functions in radian units); `sind`, `cosd`, `tand`, `asind`, `acosd`, `atand` (as before but in degree units); `norm` ($\sqrt{a^2 + b^2}$ as an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

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









































B. Named colors

 gray.black	 cmyk.WildStrawberry	 cmyk.Cerulean
 gray.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 gradients



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>linewidth.THIN</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.normal</code> (default)		<code>earrow.Large</code>
	<code>linewidth.thick</code>		<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>		