Some frequently and not so frequently asked questions about PyX

Release 0.16

Gert-Ludwig Ingold

2022/10/16

CONTENTS

1 General aspects of PyX **3**
- 1.1 The name of the game . . . 3
- 1.2 Where do I get the latest version of PyX? . . . 3
- 1.3 How can I determine the version of PyX running on my machine? . . . 3
- 1.4 How can I access older versions of PyX? . . . 3
- 1.5 Does PyX run under my favorite operating system? . . . 4
- 1.6 Under which versions of Python will PyX run? . . . 4
- 1.7 Does PyX provide a GUI to view the produced image? . . . 4
- 1.8 Will I be able to embed PyX graphics output into an IPython notebook? . . . 5
- 1.9 I am a Gnuplot user and want to try PyX. Where can I get some help? . . . 5
- 1.10 Where can I get help if my question is not answered in this FAQ? . . . 5

2 Python **7**
- 2.1 What is Python? . . . 7
- 2.2 Where can I learn more about Python? . . . 7
- 2.3 What do I need to import in order to use PyX? . . . 7
- 2.4 What is a raw string and why should I know about it when using PyX? . . . 8

3 General aspects of plotting with PyX **9**
- 3.1 How do I generate multipage output? . . . 9

4 Plotting of graphs **11**
- 4.1 General aspects . . . 11
- 4.2 Axis properties . . . 13
- 4.3 Data properties . . . 14

5 Other plotting tasks **17**
- 5.1 How can I rotate text? . . . 17
- 5.2 How can I clip a canvas? . . . 17

6 TeX and LaTeX **19**
- 6.1 General aspects . . . 19
- 6.2 TeX and LaTeX commands useful for PyX . . . 20
- 6.3 TeX and LaTeX errors . . . 20
- 6.4 Fonts . . . 22

7 Text output without TeX **23**
- 7.1 General aspects . . . 23

Acknowledgements

The following persons have in one way or the other, e.g. by asking good questions or providing answers, contributed to this FAQ: Walter Brisken, Alejandro Gaita-Arinyo, Pierre Joyot, Jörg Lehmann, John Owens, Michael Schindler, Gerhard Schmid, André Wobst.

CHAPTER

ONE

GENERAL ASPECTS OF PYX

1.1 The name of the game

Originally, the name PyX was constructed as a combination of **P**ostscript, i.e. the first output format supported by PyX, P**y**thon, i.e. the language in which PyX is written, and Te**X**, i.e. the program which PyX uses for typesetting purposes. Actually, the title of this question is a tribute to TeX because it is taken from the first chapter of the TeX book[1] where the origin of the name TeX and its pronunciation are explained.

Despite the ties between TeX and PyX, their pronunciation is quite different. According to the developers of PyX, it should be pronounced as [pʏks]. Please do not pronounce it as [pʏx] or [pʏç].

1.2 Where do I get the latest version of PyX?

The current release of PyX (as well as older ones) is freely available from https://pyx-project.org where also a subversion repository with the latest patches can be found. In addition, PyX is registered on the Python Package Index at https://pypi.org/project/PyX and can be installed by `easy_install` and `pip`. As PyX is hosted on PyPI, it can be directly downloaded and installed by `pip`. Please see the pip documentation for details.

Possibly older versions of PyX are also available as package for various Linux distributions: see, for instance, https://packages.debian.org/stable/python3-pyx for information on the PyX package in Debian GNU/Linux, https://packages.ubuntu.com/jammy/python3-pyx for Ubuntu, https://software.opensuse.org/package/python-PyX for OpenSUSE, and https://archlinux.org/packages/community/any/python-pyx for Arch Linux.

PyX has no dependencies on other Python packages.

1.3 How can I determine the version of PyX running on my machine?

Start a python session (usually by typing `python` at the system prompt) and then type the following two commands (>>> is the python prompt)

```
>>> import pyx
>>> pyx.__version__
```

Note that there are two underscores before and after `version`.

1.4 How can I access older versions of PyX?

There are reasons which might make it necessary to use older versions of PyX. If you are using Python 2 you will need PyX version 0.12.1 or earlier (see *Under which versions of Python will PyX run?*). Furthermore, as at present it is not guaranteed that PyX is backward compatible, it may be desirable to access an older version of PyX instead of adapting older code to a more recent version of PyX. In order to do that, one needs the corresponding PyX package (see *Where do I get the latest version of PyX?* if you need to download it), which should be unpacked

[1] D.Knuth, *The TeX book* (Addison-Wesley, 1984)

below a directory, e.g. `/home/xyz/Python`, where you want to keep the various PyX versions. This will result in a subdirectory with a name like `PyX-0.16` which contains the contents of the corresponding package. You can then ask Python to first look in the appropriate directory before looking for the current version of PyX by inserting the following code (appropriately modified according to your needs) at the beginning of your program before importing the PyX module:

```
import sys
sys.path.insert(0, "/home/xyz/Python/PyX-0.16")
```

Including appropriate lines even if the current version of PyX is used, might turn out to be helpful when the current version has become an old version (unless you have no difficulties determining the PyX version by looking at your code).

If your operating system supports path expansion, you might use as an alternative:

```
import sys, os
sys.path.insert(0, os.path.expanduser("~/Python/PyX-0.16"))
```

which will expand the tilde to your home directory.

1.5 Does PyX run under my favorite operating system?

Yes, if you have installed Python (*What is Python?*) and TeX (*What is TeX/LaTeX?*). Both are available for a large variety of operating systems so chances are pretty good that you will get PyX to work on your system.

1.6 Under which versions of Python will PyX run?

Starting with version 0.13, PyX requires Python 3.2 or higher. If you still need to run PyX with Python 2, you should use version 0.12.1 which is designed to run with Python 2.3 up to 2.7.

The version of your Python interpreter can be determined by calling it with the option `-V`. Alternatively, you can simply start the interpreter and take a look at the startup message. Note that there may be different versions of Python installed on your system at the same time. The default Python version need not be the same for all users.

1.7 Does PyX provide a GUI to view the produced image?

No, PyX itself does not provide a means to view the produced image. The result of a PyX run is an EPS (= Encapsulated PostScript) file, a PS (= PostScript) file, a PDF (= Portable Document Format) file or a SVG (= Scalable Vector Graphics) file, which can be viewed, printed or imported into other applications.

There are several means of viewing PS and EPS files. A common way would be to use `ghostview` which provides a user interface to the PostScript interpreter `ghostscript`. More information about this software, which is available for a variety of platforms, can be found at http://www.cs.wisc.edu/~ghost/. If you do not own a printer which is capable of printing PostScript files directly, `ghostscript` may also be useful to translate PS and EPS files produced by PyX into something your printer will understand.

PDF files can be viewed by means of the `Adobe Reader ®` available from http://www.adobe.com/products/acrobat/readstep2.html. On systems running X11, `xpdf` might be an alternative. It is available from http://www.foolabs.com/xpdf/.

SVG files can be viewed by webbrowsers like Firefox available at https://www.mozilla.org/en-US/firefox or Chrome available at https://www.google.com/chrome/.

If you want to do interactive development of a PyX graphics, you might consider to use an IPython notebook (see *Will I be able to embed PyX graphics output into an IPython notebook?*).

1.8 Will I be able to embed PyX graphics output into an IPython notebook?

Yes, PyX canvas object and objects inheriting from the canvas class, in particular graphs and text, can be embedded into an IPython notebook. Suppose you have a canvas object called `c` on which you have done some drawing. Then entering `c` in an IPython notebook cell and executing that cell will automatically produce a SVG representation and embed it into the notebook. (Alternatively, also PNG is available by means of ghostscript, but the default display_order of IPython prefers SVG over PNG.) For more information on IPython and working with its notebooks see http://www.ipython.org/.

1.9 I am a Gnuplot user and want to try PyX. Where can I get some help?

There exists a tutorial by Titus Winters which explains how to perform standard Gnuplot tasks with PyX. The tutorial can be found at http://www.cs.ucr.edu/~titus/pyxTutorial/.

1.10 Where can I get help if my question is not answered in this FAQ?

The PyX sources contain a reference manual which is also available online at https://pyx-project.org/manual/. Furthermore, there exists a set of examples demonstrating various features of PyX, which is available in the sources or can be browsed at https://pyx-project.org/examples.html. If the feature you are looking for is among them, using the appropriate part of the example code or adapting it for your purposes may help.

There is also a user discussion list about PyX which you can subscribe to at http://lists.sourceforge.net/lists/listinfo/pyx-user. The archive of the discussion list is available at http://sourceforge.net/mailarchive/forum.php?forum_name=pyx-user.

Finally, it might be worth checking https://pyx-project.org/pyxfaq.pdf for an updated version of this FAQ.

CHAPTER

TWO

PYTHON

2.1 What is Python?

From www.python.org:

> Python is an *interpreted, interactive, object-oriented* programming language. It is often compared to Tcl, Perl, Scheme or Java.
>
> Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries, as well as to various windowing systems (X11, Motif, Tk, Mac, MFC). New built-in modules are easily written in C or C++. Python is also usable as an extension language for applications that need a programmable interface.
>
> The Python implementation is portable: it runs on many brands of UNIX, on Windows, OS/2, Mac, Amiga, and many other platforms. If your favorite system isn't listed here, it may still be supported, if there's a C compiler for it. Ask around on comp.lang.python – or just try compiling Python yourself.
>
> The Python implementation is copyrighted but **freely usable and distributable, even for commercial use**.

2.2 Where can I learn more about Python?

The place to start is www.python.org where you will find plenty of information on Python including tutorials.

2.3 What do I need to import in order to use PyX?

It is recommended to begin your Python code with:

```
from pyx import *
```

when using PyX. This allows you for example to write simply `graph.graphxy` instead of `pyx.graph.graphxy`. The following modules will be loaded: `attr`, `box`, `bitmap`, `canvas`, `color`, `connector`, `deco`, `deformer`, `document`, `epsfile`, `graph`, `path`, `pattern`, `style`, `trafo`, `text`, and `unit`.

For convenience, you might import specific objects of a module like in:

```
from graph import graphxy
```

which allows you to write `graphxy()` instead of `graph.graphxy()`.

All code segments in this document assume that the import line mentioned in the first code snippet is present.

2.4 What is a raw string and why should I know about it when using PyX?

The backslash serves in standard Python strings to start an escape sequence. For example `\n` corresponds to a newline character. On the other hand, TeX and LaTeX, which do the typesetting in PyX, use the backslash to indicate the start of a command. In order to avoid the standard interpretation, the string should be marked as a raw string by prepending it by an `r` like in:

```
c.text(0, 0, r"$\alpha\beta\gamma$")
```

CHAPTER

THREE

GENERAL ASPECTS OF PLOTTING WITH PYX

3.1 How do I generate multipage output?

With versions 0.8 and higher it is possible to produce multipage output, i.e. a Postscript or PDF file containing more than one page. In order to achieve this, one creates pages by drawing on a canvas as usual and appends them in the desired order to a document from which Postscript or PDF output is produced. The following example serves as an illustration:

```
from pyx import *

d = document.document()
for i in range(3):
    c = canvas.canvas()
    c.text(0, 0, "page %i" % (i+1))
    d.append(document.page(c, paperformat=document.paperformat.A4,
                               margin=3*unit.t\_cm,
                               fittosize=1))
d.writePSfile("multipage")
```

Here, d is the document into which pages are inserted by means of the append method. When converting from a canvas to a document page, the page properties like the paperformat are specified. In the last line, output is produced from document d.

CHAPTER

FOUR

PLOTTING OF GRAPHS

4.1 General aspects

4.1.1 How do I generate a graph from data as simply as possible?

Suppose that you have a data file `x.dat` containing values for `x` and `y` in two columns. Then the following code will do the job:

```
from pyx import *

g = graph.graphxy(width=10)
g.plot(graph.data.file("x.dat", x=1, y=2))
g.writeEPSfile("x")
```

`graphxy` creates a canvas (called `g` in this example) onto which the graph will be drawn and it sets the default behavior including the axis. There is, however, no default value for the width of the graph. In `plot` you have to specify the name of the data file and the columns from which the data should be taken. Finally, `writeEPSfile` will generate the postscript file `x.eps` which you can view or print.

A minimal example is also provided in the PyX distribution as `examples/graphs/minimal.py`.

4.1.2 How do I generate a graph of a function as simply as possible?

The following example will draw a parabola:

```
from pyx import *

g = graph.graphxy(width=10,
                  x=graph.axis.linear(min=-2, max=2)
                  )

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

g.writeEPSfile("x")
```

Most of the code has been explained in *How do I generate a graph from data as simply as possible?*. The main difference is that here you need to specify minimum and maximum for the `x`-axis so that PyX knows in which range to evaluate the function.

Another, slightly more complex, example is also provided in the PyX distribution as `examples/graphs/piaxis.py`.

4.1.3 How can I stack graphs?

PyX always needs a canvas to draw on. One possibility therefore consists in creating a new canvas with

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

and inserting the graphs into this canvas with `c.insert(...)`. Here, `...` has to be replaced by the name of the graph. Alternatively, the canvas created with `graph.graphxy` for one of the graphs can be used to insert the other graphs even if they will be positioned outside the first graph.

The second issue to address is positioning of the graphs. By specifying `xpos` and `ypos` when calling `graphxy` you can define the position of a graph. Later on, the position and size of a graph `g` can be referred to as `g.xpos` `g.ypos` `g.width` and `g.height` even if for example the height has never been specified explicitly but is only defined by a PyX default.

The following example shows how to put graph `gupper` above graph `glower` on a canvas `c`:

```
from pyx import *
from graph import graphxy

c = canvas.canvas()

glower = graphxy(width=10)
glower.plot(...)
c.insert(glower)

gupper = graphxy(width=10, ypos=glower.ypos+glower.height+2)
gupper.plot(...)

c.insert(gupper)
c.writeEPSfile(...)
```

where `...` has to be replaced by the appropriate information like data and symbol specifications and the name of the output file. Here, `c.insert` is used to actually insert the subcanvasses for the graphs into the main canvas `c` and `c.writeEPSfile` in the last line requests to write the contents of this canvas to a file.

4.1.4 How can I plot grid data?

PyX offers support for plotting three-dimensional data as two-dimensional color plots or grey-scale plots and of vector fields by providing ways to plot rectangles and arrows in graphs.

We start by considering the task of creating a two-dimensional color plot by plotting a number of filled rectangles. One first needs to create a data set which consists of five entries per data point. These are the lower left corner $(x_{\min}, y_{\min})$ and the upper right corner $(x_{\max}, y_{\max})$ of the triangle and a value between 0 and 1 determining the color via a PyX color palette. The following code gives an idea of how to proceed:

```
g.plot(graph.data.file("datafile.dat", xmin=1, xmax=2, ymin=3, ymax=4, color=5),
               [graph.style.rect(color.palette.ReverseRainbow)]
        )
g.dodata()
```

Here, we assume that the data are stored in `datafile.dat` and the columns contain $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$, and the color value in this order. The columns are numbered from 1, since the 0th column contains the line number. To determine the color, we use the `ReverseRainbow` palette. The last line instructs PyX to plot the rectangles before plotting the axes. Otherwise, the axes might be covered partially by the rectangles and, in particular, the ticks might not be visible. Gray-scale plots can easily be generated by specifying the palette `Gray` or `ReverseGray` (cf. appendix C of the manual for a list of predefined palettes).

At first sight, it seems surprising that plotting of grid data requires the specification of four coordinates for the rectangle. The reason is that this allows to draw rectangles of varying sizes which may help to reduce the size of the postscript file by combining rectangles of the same color in horizontal or vertical direction. For example, it may be sufficient to plot a grey-scale image in a small number of grey shades and then combining rectangles may be appropriate. Note, though, that this step is part of the data creation and not preformed by PyX. Another advantage of fully specifying each rectangle is that it is straightforward to leave parts of the graph blank.

The same ideas as for the color plot can be applied to plot vector fields where each data point is represented by an arrow. In this case a data point is specified by the position of the arrow, its size and its direction as indicated in the

following code snippet:

```
g.plot(graph.data.file("datafile.dat"), x=1, y=2, size=3, angle=4),
           [graph.style.arrow()]
       )
```

Complete code examples can be found in `examples/graphs/mandel.py` and `examples/graphs/arrows.py` .

4.1.5 How can I access points in problem coordinates of a graph?

Sometimes it may be necessary to add graphical elements to a graph in addition to the data or function(s) which have been plotted as described in *How do I generate a graph from data as simply as possible?* and *How do I generate a graph of a function as simply as possible?*. For a graph instance `g` the positioning can easily be done in canvas coordinates by making use of the origin (`g.xpos`, `g.ypos`) and the width (`g.width`) and height (`g.height`) of the graph.

Occasionally, it may be more convenient to specify the position of the additional material in terms of problem coordinates. However, this requires that the mapping from problem coordinates to canvas coordinates is known. By default this is not the case before the content of the canvas is written to the output which is too late for our purpose. One therefore needs to explicitly instruct PyX to determine this mapping. One possibility is to ask PyX to finish the graph by means of `g.finish()`. Now, problem coordinates can be used to insert additional material which will end up in front of the graph. If this is not desired, one should only fix the layout of the graph by means of `g.dolayout()`. Then, the additional material can be put onto the canvas before the graph is drawn and it will therefore appear behind the graph.

The conversion of problem coordinates (`px`, `py`) to canvas coordinates (`x`, `y`) is performed as follows:

```
x, y = g.pos(px, py)
```

By default, the problem coordinates will refer to the ranges of the x and y axes. If several axes with different ranges exist, the instances of the desired axes should be passed to the `pos` method by means of the keyword arguments `xaxis` and `yaxis`.

We remark that the drawing of lines parallel to one of the axes at specific problem coordinates can also be done by adapting the method described in *How do I plot the zero line?*.

4.1.6 I would like a key for only some of my data sets. How do I do that?

Todo

This still needs to be answered.

4.2 Axis properties

4.2.1 How do I specify the tick increment?

In the partition of a linear axis, the increments associated with ticks, subticks etc. can be specified as argument of `parter.linear`. In the following example, ticks will be drawn at even values while subticks will be drawn at all integers:

```
from pyx.graph import axis

tg = graph.graphxy(width=10,
                   x=axis.linear(min=1, max=10,
                                 parter=axis.parter.linear(tickdists=[2,1]))
                   )
```

4.2.2 How do I plot the zero line?

PyX releases before 0.6 offered the possibility to stroke a zero line by specifying `zeropathattrs` in the painter constructor. In more recent releases, one proceeds as follows. First one has to fix the layout information of the graph by means of the `finish` or `dolayout` method (see *How can I access points in problem coordinates of a graph?* for a more detailed explanation). Then, the `xgridpath` or `ygridpath` method of a graph will return a grid path parallel to the y or x axis, respectively, at the specified y value. As an example, a zero line in x direction can be drawn as follows:

```
g.finish()
g.stroke(g.ygridpath(0))
```

4.2.3 How can I add grid lines to a graph?

Specifying `gridattrs` for the painter of an axis will generate grid lines orthogonal to this axis. At least an empty list is needed like in

```
g = graph.graphxy(width=10,
                  x=graph.axis.linear(painter=graph.axis.painter.
↪regular(gridattrs=[])),
                  y=graph.axis.linear()
                  )
```

where grid lines in vertical direction are drawn in default style.

Occassionally, one might want to draw grid lines corresponding to ticks and subticks in a different style. This can be achieved by specifiying changeable attributes using `changelist`. The following code

```
my_xpainter = graph.axis.painter.regular(gridattrs=
                    [attr.changelist([style.linestyle.solid, style.linestyle.dashed])]
                                        )
my_ypainter = graph.axis.painter.regular(gridattrs=
                    [attr.changelist([color.rgb.red, color.rgb.blue])]
                                        )

g = graph.graphxy(width=10,
                  x=graph.axis.linear(painter=my_xpainter),
                  y=graph.axis.linear(painter=my_ypainter)
                  )
```

will create vertical solid and dashed grid lines for ticks and subticks, respectively. The horizontal grid lines will be red for ticks and blue for subticks. The changeable attributes are applied in a cyclic manner. Therefore, in this example grid lines at subsubticks would be plotted in the same style as for ticks. If this is not desired, the list of attributes should be extended by an appropriate third style. The keyword `None` will switch off the respective level of grid lines in case you want to draw them only e.g. for ticks but not subticks.

4.3 Data properties

4.3.1 How do I choose the symbol and its attributes?

Suppose a graph called `g` has been initialized, e.g. by using `graph.graphxy`. Then, data and the style of their representation in the graph are defined by calling `g.plot` like in the following example in which filled circles are requested:

```
g.plot(graph.data.file("test.dat"),
       [graph.style.symbol(graph.style.symbol.circle, symbolattrs=[deco.filled])]
       )
```

As another example, if the linewidth of the symbol is too thin for your purposes, you could use something like:

```
[graph.style.symbol(graph.style.symbol.plus, symbolattrs=[style.linewidth.Thick])]
```

4.3.2 How do I choose the color of the symbols?

Colors are not properties of the symbol as such and can therefore not be specified in `symbolattrs` directly. The color is rather related to the plotting of the symbol as defined by `deco.stroked` or `deco.filled`. With

```
graph.style.symbol(graph.style.symbol.circle,
                   symbolattrs=[deco.stroked([color.rgb.red]),
                                deco.filled([color.rgb.green])]
                   )
```

you will obtain a circle filled in green with a red borderline.

4.3.3 How do I choose the line style?

If you do not want to use symbols, you can set the line style as in this example

```
g.plot(graph.data.file("test.dat"),
       [graph.style.line([style.linewidth.Thin])]
       )
```

where the linewidth is set.

If you also want to use symbols, you can combine the symbol and the line style as in

```
g.plot(graph.data.file("test.dat"),
       [graph.style.line(lineattrs=[style.linewidth.Thin,
                                    style.linestyle.dashed]),
        graph.style.symbol(graph.style.symbolline.circle,
                           symbolattrs=[deco.filled])
        ]
       )
```

to plot the symbols on top of a thin, dashed line. You may alter the order of the styles to plot the line on top of the symbols.

4.3.4 How can I change the color of symbols or lines according to a palette?

If several data sets should be plotted in different colors, one can specify in `symbolattrs` and/or `lineattrs` a palette like `color.palette.Rainbow`. Equidistant colors are chosen spanning the palette from one end to the other. For example, for three data sets the colors are chosen from the palette at 0, 0.5, and 1. For the rainbow palette, this would correspond to red, green, and blue, respectively.

In the following example, symbols vary in form and change their color according to the rainbow palette at the same time as the connecting lines:

```
mystyle = [graph.style.symbol(graph.style.symbol.changecircle,
                              symbolattrs=[color.palette.Rainbow]),
           graph.style.line(lineattrs=[color.palette.Rainbow])]
```

See question *How can I specify changing colors (or other attributes) for symbols or lines?* for a more complete example demonstrating how to use this style definition and for a comment on the necessity of defining `mystyle` (you are of course free to choose a different name).

4.3.5 How can I specify changing colors (or other attributes) for symbols or lines?

In `symbolattrs` and/or `lineattrs` so-called changelist can be used. As an example

```
mystyle = graph.style.symbol(symbolattrs=
                             [attr.changelist([color.rgb.red, color.rgb.green])])
g.plot(graph.data.file("x.dat", x=1, y=2), [mystyle])
g.plot(graph.data.file("y.dat", x=1, y=2), [mystyle])
g.plot(graph.data.file("z.dat", x=1, y=2), [mystyle])
```

will switch between red and green symbols each time a new data set is plotted. Several changelists can be specified. They are cycled independently and need not be of the same length. It should be noted that the definition of `mystyle` in this example ensures that there is only one instance of the definition of `symbolattrs`. Putting an explicit definition of `symbolattrs` in each call to `plot` would not lead to the desired result because each time a new instance would be created which then starts with the first item in the changelist.

It may be necessary to repeat attributes in order that several changelists cooperate to produce the desired result. A common situation is that one would like to cycle through a list of symbols which should be used in alternating colors. This can be achieved with the following code:

```
mystyle = graph.style.symbol(
                graph.style.symbol.changetriangletwice,
                symbolattrs=[attr.changelist([color.rgb.red, color.rgb.green])])
```

which will produce a red triangle, a green triangle, a red circle, a green circle and so on for diamond and square because `changetriangletwice` lists each symbol twice. If instead of changing between colors one would like to change between filled and open symbols, one can make use of a predefined changelist

```
mystyle = graph.style.symbol(
                graph.style.symbol.changetriangletwice,
                symbolattrs=[graph.style.symbol.changefilledstroked])
```

CHAPTER

FIVE

OTHER PLOTTING TASKS

5.1 How can I rotate text?

Text can be written at an arbitrary angle by specifying the appropriate transformation as an attribute. The command

```
c.text(0, 0, "Text", [trafo.rotate(60)])
```

will write at an angle of 60 degrees relative to the horizontal axis. If no pivot is specified (like in this example), the text is rotated around the reference point given in the first two arguments of `text`. In the following example, the pivot coincides with the center of the text:

```
c.text(0, 0, "Text", [text.halign.center,text.valign.middle,trafo.rotate(60)])
```

5.2 How can I clip a canvas?

In order to use only a part of a larger canvas, one may want to clip it. This can be done by creating a clipping object which is used when creating a canvas instance:

```
clippath = path.circle(0.,0.,1.)
clipobject = canvas.clip(clippath)
c = canvas.canvas([clipobject])
```

In this example, the clipping path used to define the clipping object is a circle.

CHAPTER

SIX

TEX AND LATEX

6.1 General aspects

6.1.1 What is TeX/LaTeX?

TeX is a high quality typesetting system developed by Donald E. Knuth which is available for a wide variety of operating systems. LaTeX is a macro package originally developed by Leslie Lamport which makes life with TeX easier, in particular for complex typesetting tasks. The current version of LaTeX is referred to as LaTeX2e and offers e.g. improved font selection as compared to the long outdated LaTeX 2.09 which should no longer be used.

6.1.2 Do I need to use TeX/LaTeX?

Traditionally, typesetting in PyX is done with TeX or LaTeX. With PyX 0.15, the Unicode engine was introduced. This engine can be useful if one does not need the elaborate typesetting capabilities of TeX and LaTeX and wants to avoid installing the corresponding packages. A TeX installation may nevertheless be useful if one wants to make use of fonts distributed with it. For further information see *Text output without TeX*.

6.1.3 I don't know anything about TeX and LaTeX. Where can I read something about it?

Take a look at CTAN (cf. *What is CTAN?*) where in CTAN:info you may be able to find some useful information. There exists for example "A Gentle Introduction to TeX" by M. Doob (CTAN:gentle/gentle.pdf) and "The Not So Short Introduction to LaTeX2e" (CTAN:info/lshort/english/lshort.pdf) by T. Oetiker et al. The latter has been translated into a variety of languages among them korean (which you will not be able to read unless you have appropriate fonts installed) and mongolian.

Of course, it is likely that these documents will go way beyond what you will need for generating graphics with PyX so you don't have to read all of it (unless you want to use TeX or LaTeX for typesetting which can be highly recommended).

There exists also a number of FAQs on TeX at CTAN:help.

6.1.4 What is CTAN?

CTAN is the *Comprehensive TeX Archive Network* where you will find almost everything related to TeX and friends. The main CTAN server is www.ctan.org but there exists a large number of mirrors around the world. You can help to reduce the load on the main server by using mirror.ctan.org which will redirect you to a mirror nearby. A list of known mirrors is available at http://mirror.ctan.org/README.mirrors.

In this FAQ, `CTAN:` refers to the root of the CTAN tree, e.g. http://www.ctan.org/tex-archive/. The links to CTAN in this document point to the main server but you might consider using a server closer to you in order to reduce traffic load.

6.1.5 Is there support for ConTeXt?

No, and as far as I know there no plans to provide it in the near future. Given the close ties between ConTeXt and MetaPost, ConTeXt users probably prefer to stick with the latter anyway.

6.2 TeX and LaTeX commands useful for PyX

6.2.1 How do I get a specific symbol with TeX or LaTeX?

A list of mathematical symbols together with the appropriate command name can be found at CTAN:info/symbols/math/symbols.pdf. A comprehensive list containing almost 6000 symbols for use with LaTeX can be obtained from CTAN:info/symbols/comprehensive/symbols-a4.pdf. In some cases it might be necessary to install fonts or packages available from CTAN (cf. *What is CTAN?*).

6.3 TeX and LaTeX errors

6.3.1 Undefined control sequence `\usepackage`

The command `\usepackage` is specific to LaTeX. Since by default PyX uses TeX, you have to specify the correct typesetting engine:

```
text.set(text.LatexEngine)
```

6.3.2 Undefined control sequence `\frac`

The command `\frac` is only available in LaTeX. The equivalent to `\frac{a}{b}` in TeX is `{a \over b}`. As an alternative you may ask for the LaTeX engine as explained in *Undefined control sequence \usepackage*.

6.3.3 Missing `$` inserted

You have specified TeX- or LaTeX-code which is only valid in math mode. Typical examples are greek symbols, sub- and superscripts or fractions.

On the PyX level, you can specify math mode for the whole string by using `text.mathmode` as in

```
c.text(0, 0, r"{\alpha}", text.mathmode)
```

Keep also in mind that the standard Python interpretation of the backslash as introducing escape sequences needs to be prevented.

On the TeX/LaTeX level you should enclose the commands requiring math mode in `$`'s. As an example, `$\alpha_i^j$` will produce a greek letter alpha with a subscript i and a superscript j. The dollar sign thus allows you to specify math mode also for substrings. There exist other ways to specify math mode in TeX and LaTeX which are particularly useful for more complex typesetting tasks. To learn more about it, you should consult the documentation *I don't know anything about TeX and LaTeX. Where can I read something about it?*.

6.3.4 Why do environments like `itemize` or `eqnarray` seem not to work?

An itemize environment might result in a LaTeX error complaining about a `missing \item` or an eqnarray might lead to a LaTeX message `missing \endgroup inserted` even though the syntax appears to be correct. The TeXnical reason is that in PyX text is typeset in left-right mode (LR mode) which does not allow linebreaks to occur. There are two ways out.

If the text material should go in a box of given width, a parbox can be used like in the following example:

```
text.set(text.LatexEngine)
c = canvas.canvas()
w = 2
c.text(0, 0, r"\begin{itemize}\item a\item b\end{itemize}", [text.parbox(w)])
```

Occasionally, one would like to have the box in which the text appears to be as small as possible. Then the `fancybox` package developed by Timothy Van Zandt is useful which provides several environments like `Bitemize` and `Beqnarray` which can be processed in LR mode. The relevant part of the code could look like:

```
text.set(text.LatexEngine)
text.preamble(r"\usepackage{fancybox}")
c = canvas.canvas()
c.text(0, 0, r"\begin{Bitemize}\item a\item b\end{Bitemize}")
```

Other environments provided by the `fancybox` package include `Bcenter`, `Bflushleft`, `Bflushright`, `Benumerate`, and `Bdescription`. For more details, the documentation of the package should be consulted.

6.3.5 Font shape `OT1/xyz/m/n` undefined

You have asked to use font `xyz` which is not available. Make sure that you have this font available in Type1 format, i.e. there should be a file `xyz.pfb` somewhere. If your TeX system is TDS compliant (TDS=TeX directory structure, cf. CTAN:tds/draft-standard/tds/tds.pdf) you should take a look at the subdirectories of `$TEXMF/fonts/type1`.

6.3.6 File . . . is not available or not readable

Such an error message might already occur when running the example file `hello.py` included in the PyX distribution. Usually, the error occurs due to an overly restrictive umask setting applied when unpacking the `tar.gz` sources. This may render the file mentioned in the error message unreadable because the python distutil installation package doesn't change the file permissions back to readable for everyone.

If the file exists, the problem can be solved by changing the permissions to allow read access.

6.3.7 No information for font `cmr10` found in font mapping file

Such an error message can already be encountered by simply running the example file `hello.py` included in the PyX distribution. The likely reason is that the TeX system does not find the cmr fonts in Type1 format. PyX depends on these fonts as it does not work with the traditional pk fonts which are stored as bitmaps.

Therefore, the first thing to make sure is that the cmr Type1 fonts are installed. In some TeX installations, the command `kpsewhich cmr10.pfb` will return the appropriate path if the cmr fonts exist in the binary Type1 format (extension `pfb`) required by PyX. If the command does not work but the TeX system is TDS compliant (*Font shape OT1/xyz/m/n undefined*), a look should be taken at `$TEXMF/fonts/type1/bluesky/cm` where `$TEXMF` is the root of the `texmf` tree.

If the Type1 fonts do not exist on the system, they may be obtained from the CTAN (cf. *What is CTAN?*) at CTAN:fonts/cm/ps-type1/bluesky). See the `README` for information about who produced these fonts and why they are freely available.

If the Type1 fonts exist, the next step is to take a look at `psfonts.map`. There may be several files with this name on the system, so it is important to find out which one TeX is actually using. `kpsewhich psfonts.map` might give this information.

The most likely problem is that this file does not contain a line telling TeX what to do if it encounters a request for font `cmr10`, i.e. the following line may be missing

```
cmr10           CMR10           <cmr10.pfb
```

It is probable that the required lines (in practice, you do not just need `cmr10`) are found in a file named `psfonts.cmz` which resides in `$TEXMF/dvips/bluesky`.

One solution is to instruct PyX to read additional map files like `psfonts.cmz` or `psfonts.amz`. This can be achieved by modifying the appropriate `pyxrc` file which is either the systemwide `/etc/pyxrc` or `.pyxrc` in the user's home directory. Here, the names of the map files to be read by PyX should be appended separated by whitespaces like in the following example:

```
[text]
fontmaps = psfonts.map psfonts.cmz psfonts.amz
```

The same effect can be achieved by inserting the following line into the PyX code:

```
text.set(fontmaps="psfonts.map psfonts.cmz psfonts.amz")
```

Note that the default map (`psfonts.map`) has to be specified explicitly.

An alternative approach consists in modifying the TeX installation by inserting the contents of the desired map files like `psfonts.cmz` into `psfonts.map`. Probably, `psfonts.map` recommends not to do this by hand. In this case the instructions given in the file should be followed. Otherwise, `psfonts.cmz` should be copied into `psfonts.map` while keeping a backup of the old `psfonts.map` just in case. After these changes, PyX most likely will be happy. When inserting `psfonts.cmz` into `psfonts.map` it may be a good idea to include `psfonts.amz` as well. `psfonts.amz` contains information about some more fonts which might be needed at some point. Making these changes to `psfonts.map` will imply that the TeX system will use the cmr fonts in Type1 format instead of pk format which is actually not a bad thing, in particular if `latex / dvips / ps2pdf` is used to generate PDF output. With fonts in pk format this will look ugly and using Type1 fonts solves this problem as well. When `pdflatex` is used to create PDF files, Type1 fonts will be used anyway.

6.4 Fonts

6.4.1 I have Type1 fonts in `pfa` format. How do I obtain the corresponding `pfb` files needed by PyX?

6.4.2 I want to use a font other than computer modern roman

As long as you have a font in Type1 format available, this should be no problem (even though it may cost you some time to set up things properly).

In the simplest case, your LaTeX system contains everything needed. Including the following line into your code will probably work:

```
text.set(text.LatexEngine)
text.preamble(r"\usepackage{mathptmx}")
```

and give you Times as roman font.

If you own one of the more common commercial fonts, take a look at CTAN:fonts and its subdirectories as well as at the web page http://home.vr-web.de/was/fonts.html of Walter Schmidt. It is not unlikely that somebody has already done most of the work for you and created the files needed for the font to work properly with LaTeX. But remember: we are talking about commercial fonts here, so do not expect to find the fonts themselves for free.

If none of these cases applies, you should spend some time reading manuals about font installation, e.g. CTAN:macros/latex/doc/fntguide.pdf (of course, I do not expect font wizards to read the last few lines).

6.4.3 Can I use a TrueType font with PyX?

Not directly as PyX only knows how to handle Type1 fonts (although it is possible to get LaTeX to work with TrueType fonts). However, you may use `ttf2pt1` (from ttf2pt1.sourceforge.net) to convert a TrueType font into a Type1 font which you then install in your TeX system (cf. *I want to use a font other than computer modern roman*). You will loose hinting information in the conversion process but this should not really matter on output devices with not too low resolution.

CHAPTER

SEVEN

TEXT OUTPUT WITHOUT TEX

7.1 General aspects

7.1.1 How can I typeset text in PyX without TeX?

Starting with PyX 0.15, an additional typesetting engine besides the TeX and LaTeX engine is available: the Unicode engine. It can be made the default typesetting engine by:

```
text.set(text.UnicodeEngine)
```

The font and its size can be specified through additional parameters like in this example:

```
text.set(text.UnicodeEngine, fontname='cmss10', size=20)
```

7.1.2 Do I still need a TeX installation?

Not necessarily. However, PyX needs to have access to the Type1 fonts it is supposed to use in the typesetting. Specifically, it needs the corresponding `pfb` and `afm` files. There are different ways to provide them.

The first possibility is indeed a TeX installation. This approach makes particularly sense if the fonts to be used are made available by a TeX distribution. For example, the Computer Modern fonts used by PyX as default fonts are provided by all TeX installations.

If, on the other hand, a font is to be used for which no TeX support is readily available, its `pfb` and `afm` files can be put in the directory where the Python script is placed. The font will then be found by PyX if the parameter `fontname` corresponds to the basename of the `pfb` and `afm` files.

The latter approach is not optimal if the font is used in different PyX scripts scattered over several directories or if a larger number of fonts is needed. Then it is better to store the Type1 font files in a central place and to tell PyX about the files as explained in the following example.

Suppose the standard TeX fonts should be used without a TeX installation. The fonts can be obtained in Type1 format from https://www.ams.org/publications/authors/tex/amsfonts. Extract the zip file somewhere on your file system and generate an index file (ls-R) by running `ls -R > ls-R` in the directory to which the fonts were extracted. Finally create a `.pyxrc` file in your home directory with the following content:

```
[filelocator]
methods = local internal ls-R
ls-R = /<the full path of the directory you extracted the amsfonts zip file>/ls-R
```

The generation of the index file can be skipped by using the `recursivedir` locator instead:

```
[filelocator]
methods = local internal recursivedir
recursivedir = /<the full path of the directory you extracted the amsfonts zip file>
```

However, the use of an index file increases performance, as PyX does not need to crawl the directory structure to locate the actual files.

7.1.3 Is it possible to use the Unicode engine in addition to the TeX engine?

Yes, it is possible to use different typesetting engines in the same script. The following example uses the Unicode engine and the TeX engine to typeset text on a canvas `c`:

```
unicode_engine = text.UnicodeEngine()
c.insert(unicode_engine.text(0, 0, "Hello, world!"))
tex_engine = text.TexEngine()
c.insert(tex_engine.text(0, 1, "Hello, world!"))
```

If needed, the font name and size can be passed as parameters to the `UnicodeEngine` constructor:

```
engine = text.UnicodeEngine(fontname='cmss10', size=20)
```