PyScript

Postscript Graphics with Python

version 0.6

Alexei Gilchrist
Paul Cochrane

0: Contents

1: Introduction

1.1. Overview

PyScript is a python package for creating high-quality postscript drawings. It began from the frustration of trying to create some good figures for publication that contained some abritrary LaTeX expressions, and has been largely inspired by *mpost*. What began as some quick-n-dirty hacks has evolved into a really useful tool (after several rewrites). Essentially a figure is scripted using python and some pre-defined objects such as rectangles, lines, text etc. This approach allows for a precise placement of all the components of a figure.

Some of the key features are

- All scripting is done in python, which is a high-level, easy to learn, well developed scripting language.
- All the objects can be translated, scaled, rotated, ... in fact any affine transformation.
- The plain text object is automatically kerned.
- You can place abritrary LaTeX expressions on your figures.
- You can create your own objects, and develop a library of figure primitives.
- Output is publication quality.

1.2. Conventions and Pitfalls

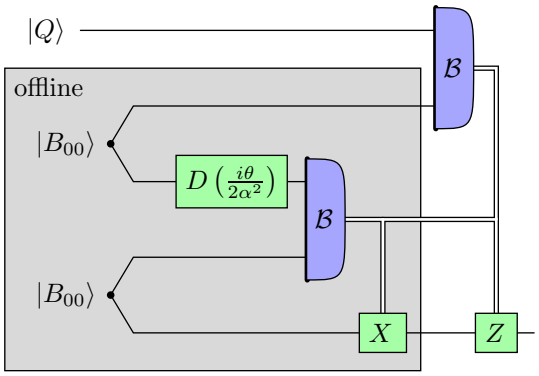
Just to be clear from the outset, some conventions follow, and some common pitfalls to be aware of ...

- The co-ordinate system is as you learned at school in maths ... the x-axis extends to the right, the y-axis extends upwards. I know, this is obvious, but a suprising number of graphics libraries invert the y-axis.
- Angles are in degrees and proceed clockwise from the top ... just like your clock. Often, key points are labeled by the compass points: n, ne, e, se, s, sw, w, nw.
- The default units are postscript points, 1cm = 28.346pp. For a figure, the default can easily be changed with the command *defaults.units=UNITS['cm']*. All of the examples in this manual are in `cm`.
- In python, an integer divided by an integer is truncated to an integer, To avoid this use floating point numbers, e.g. *2/3 = 0* but *2/3. = 0.6666*.

- Backslashes in strings have special significance, such as denoting newlines (`"\n"`). This can be frustrating for entering LaTeX expresions. You can turn off this interpretation by using raw strings: just prepend an "r" to the string e.g. `g=r"$\alpha$"`

1.3. Tutorial

As a tutorial, we'll take a detailed look at the script that created the following figure:



In the following script, we've interdispersed comments explaining what we're doing, the full script is available with the other examples and is called `tutorial.py`.

First import the *PyScript* libraries, and we'll grab some objects from `pyscript.lib.quantumcircuits` too. Most scripts would have something like this at the beginning.

```
from pyscript import *
from pyscript.lib.quantumcircuits import *
```

The default units are in postscript points. I prefer to use `cm` so switch the units here. The default units are stored in `defaults.units` which is just a number giving giving the multiplying factor compared to postscript points. `UNITS` is a dictionary of factors for some common units.

```
defaults.units=UNITS['cm']
```

There's a bunch of LaTeX macros I use often. Rather than defining them each time they're needed, well define them in the `tex_head` variable in `defaults`, which defines the start of the environment where *all* the LaTeX is processed.

```
defaults.tex_head=r"""
\documentclass{article}
\pagestyle{empty}
\usepackage{amsmath}

\newcommand{\ket}[1]{\mbox{$|#1\rangle$}}
\newcommand{\bra}[1]{\mbox{$\langle #1|$}}
\newcommand{\braket}[2]{\mbox{$\langle #1|#2\rangle$}}
\newcommand{\ketbra}[2]{\mbox{$|#1\rangle\langle #2|$}}
\newcommand{\op}[1]{\mbox{\boldmath $\hat{#1}$}}
\newcommand{\R}[3]{%
\renewcommand{\arraystretch}{.5}
$\begin{array}{@{}c@{}}{#1}\\{#2}\end{array}{#3}$
\renewcommand{\arraystretch}{1}
}
\begin{document}
"""
```

Now, define the colors of some objects here to make it easy to change them everywhere in the figure later if we need to. There are a whole variety of ways to specify a color, we'll use RGB values here.

```
blue=Color(.65,.65,1)
green=Color(.65,1,.65)
```

There's a component of the figure we'll use several times, so for convenience, define it here as a function which returns the object. A separate class would also be possible, but would involve more work. We could also have created the object and used the `copy()` method to make duplicates, but that would be clumsy.

```
def BellDet(c=P(0,0)):

    H=P(0,.8)
    W=P(.5,0)
```

`D` is a D-shaped path filled in with the blue color we defined earlier.

```
    D=Path(c+H,
           C(c+H+W),
           c+W,
           C(c-H+W),
           c-H,bg=blue,
           )
```

Now return everything as a `Group`, which will then get treated as a unit in the rest of the figure.

```
    return Group(
        Path(c-H,c+H,linewidth=2),
        D,
        TeX(r'$\mathcal{B}$',c=D.c)
        )
```

To create the big gray box, we've tweaked the parameters after examining the results so that it looks nice. The dash specification is straight from postscript.

```
offline=Rectangle(height=4,width=5.5,e=P(3.5,1.5),
                  dash='[3 ] 0',bg=Color(.85))
```

Now render the figure! What about all the other bits of the figure? Well, we'll render them on the fly since we don't need to refer to the objects again. `render` is a function that can take a variable number of arguments, we'll create some of the objects in the actual function call.

Objects are rendered in the order that they appear in the `render()` call. So, we'll put on the big gray box first, this way it'll appear to be behind everything else.

```
render(
    offline,
    TeX('offline',nw=offline.nw+P(.1,-.1)),
```

Now draw the lines, and some dots. A rough sketch on a piece of paper beforehand will really help in figuring out what the co-ordinates are for what you want to draw. You can always tweak them later.

```
    Path(P(5,0),P(-.3,0),P(-.6,.5),P(-.3,1),P(2,1)),
    Path(P(2,2),P(-.3,2),P(-.6,2.5),P(-.3,3),P(3.7,3)),
    Path(P(-1,4),P(3.7,4)),

    Dot(P(-.6,.5)),
    Dot(P(-.6,2.5)),
```

Now add a double line, notice how the central region of the line in the figure is unbroken? Can you guess how it was done?

```
classicalpath(Path(P(2.1,1.5),P(4.5,1.5),P(4.5,0)),
              Path(P(3,1.5),P(3,0)),
              Path(P(3.8,3.5),P(4.5,3.5),P(4.5,1.5)),
              ),
```

Use the function we defined earlier to add those large detectors to the figure.

```
BellDet(P(2,1.5)),
BellDet(P(3.7,3.5)),
```

Add some boxed equations to the figure. This object is from the `quantumcircuits` library, and will add a box around an abritrary object.

```
Boxed(TeX(r'$D\left(\frac{i\theta}{2\alpha^2}\right)$'),c=P(1,2),bg=green),

Boxed(TeX('$X$'),c=P(3,0),bg=green),
Boxed(TeX('$Z$'),c=P(4.5,0),bg=green),
```

Finally, add some LaTeX expressions (notice some of the macros we defined earlier), and give the filename to write the postscript to. N.B. keywords, such as `file=`, have to go after parameters in a function call.

```
TeX(r'$\ket{B_{00}}$',e=P(-.7,.5)),
TeX(r'$\ket{B_{00}}$',e=P(-.7,2.5)),
TeX(r'$\ket{Q}$',e=P(-1.1,4)),

file="tutorial.eps",
)
```

We're done. Sit back and admire the figure.

2: How Do I ...?

2.1. Aligning things

PyScript has a rich structure for aligning objects. This ranges from objects which have attributes which specify a particular point such as the nw corner of the object to functions such as *Align()* and *Distribute()* which will align and distribute a group of objects.

2.1.1. Using attributes

Certain objects (mostly those subclassed from *Area*) have named points on the object that can be read or set. *Area* defines the following compass points located on a rectangle: "n", "ne", "e", "se", "s", "sw", "w", "nw". Also the center of the area is given by "c". Reading one of these attributes will return the value of that point, and setting one of these attributes will move the object so that the named point lies on the supplied one. For example, *obj1.c=obj2.c* will align the centres of the two objects. The points returned are vectors from the origin and can be manipulated in the usual ways.

Example

This will align the centre of *obj3* so that it lies half way between the centres of *obj1* and *obj2*: *obj3.c=(obj1.c+obj1.c)/2.*.

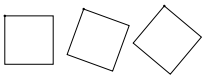
The main thing to keep in mind is that the named point is for the *objects* coordinate system. If a transformation is applied to the object, it will also be applied to all the named points.

Example

As the following example shows, the named point always stays the same in the objects coordinate system (watch the dot).

```
r=Rectangle(width=2,height=2)
g=Group()

for a in [0,20,40]:
    p=P(a/7.,0)
    r2=r.copy(c=p).rotate(a,p)
    g.append(r2,Dot(r2.nw))

render(g,file=...)
```

produces



2.1.2. Understanding boundingboxes

An objects boundingbox is a rectangle in the *current* coordinate system that completely contains the object. The bounding box for an object can be obtained with the `bbox()` method. The bounding box is calculated after all the co-ordinate transformations are applied to the objects.

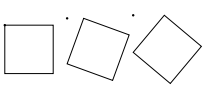
Bounding boxes have the same named point as rectangles, but these are read-only, and you can't apply transformations to bounding boxes.

Example

A variation of the previous example, where we'll put a dot at the nw corner of the bounding box

```
r=Rectangle(width=2,height=2)
g=Group()

for a in [0,20,40]:
    p=P(a/7.,0)
    r2=r.copy(c=p).rotate(a,p)
    g.append(r2,Dot(r2.bbox().nw))

render(g,file=...)
```

produces



2.1.3. Using Align()

2.1.4. Using Distribute()

2.2. Trouble Shooting

2.2.1. LaTeX Stuff

One of the useful features of *PyScript* is the ability to use LaTeX. The process is kind of complicated though so things can break. It helps to know how it all works if you're going to find some of the subtle bugs, so here's a synopsis:

1. You create some LaTeX with the *TeX()* object (you are using raw strings arent you?).
2. *PyScript* writes the text to a temporary file sandwiched between *defaults.tex_head* and *defaults.tex_tail*.
3. *defaults.tex_command* is excecuted on the file
4. *dvips* is executed on the resulting DVI file.
5. *PyScript* reads the BoundingBox comment and throws the rest away!
6. Finally, within *render* *all* the *TeX()* objects are collected together. A temporary file with all the LaTeX is generated with the individual objects delimited by postscript tags (inserted via specials) and pagebreaks.
7. As before, *defaults.tex_command* is executed (twice this time) on the file, followed by *dvips*.
8. The resulting code is parsed and divided up into sections on fonts and procedures, and the individual postscript code for each object. These are then used within the final document.

The reason it's so complicated is for efficiency — you don't want all the header and font info for every single piece of TeX you put on the page.

This is *not* the way LaTeX was designed to be used and it shows — you have to jump through a number of hoops to get it all to work. The *defaults.tex_command* should have a *-interaction=batchmode* flag or errors won't get picked up. For the same reason, *any* output from *dvips* is treated as an error, so it needs a *-q* flag. There are a number of tweaks that have to be made to the postscript code so that it is viable and the boundingboxes work ...

Despite all this it works suprisingly well. You can even include figures in the LaTeX code and input other files etc. Take care though, use *\input* rather than *\include* as the latter seems to invoke some weird things in *dvips* that result in the postscript tags not getting placed in the file. Also, don't use a figure or table environment — they're floats ... think about it.

Right, that enough of a rant. The useful stuff:

- output from the commands goes to the log file — you did look at it right?
- For each *TeX* object the temporary file thats created is called *temp1.tex*. *defaults.tex_command* and *dvips -E* execute on it to produce *temp1.eps*. All these should be valid files which you can examine. You can also run the commands by hand to see whats going on.
- The final temp file with all the objects is *temp.tex* which ends up producing *temp.ps* which will have one object per page. Again you can examine these files by hand.

2.3. Transformations and Things

3: *PyScript* Objects

These are the basic *PyScript* objects and functions. At the begining of each class there is a brief description of the structure of the class showing the relevant methods and members. See also figure 4.1 on page 22 for an indication of how the classes fit together.

3.1. Base Objects

These are classes which add layers of functionality to *PyScript* objects. Normally you wouldn't use these classes directly unless you're creating new *PyScript* objects. We'll decribe them here because they summarise what you can do with *PyScript* objects.

3.1.1. PsObj()

```
class PsObj(object):
    def __call__(self,**dict):
        Set a whole lot of attributes in one go

    def copy(self,**dict):
        return a copy of this object
        with listed attributes modified

    def __str__(self):
        return actual postscript string to generate object

    def body(self):
        subclasses should overide this for generating postscipt code

    def bbox(self):
        return objects bounding box
```

Base class of which most (all?) *PyScript* classes are subclass.

A list of parameters can be set when an object is created with calls like `t=Text('Hello',font='Helvetica')` or by calling the object like a function as in `t(sw=P(0,2))`. The parameters are also available singly as attributes: `t.sw` etc.

Printing an object produces the actual postscript code.

Objects may be copied with the `copy()` function and new parameters can be passed in as arguments eg `s = t.copy(sw=P(0,0))`.

3.1.2. AffineObj()

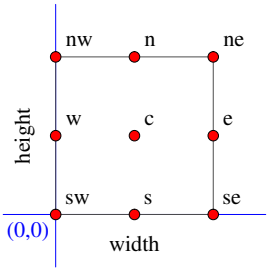
```
class AffineObj(PsObj):

    o=P(0,0)
    T=Matrix(1,0,0,1)

    def concat(self,t,p=None):
        concat matrix t to tranformation matrix
          t: a 2x2 Matrix dectribing Affine transformation
          p: the origin for the transformation
          return: reference to self

    def move(self,*args):
        translate object by a certain amount
          param args: amount to move by, can be given as
            - dx,dy
            - P
          return: reference to self

    def rotate(self,angle,p=None):
        rotate object,
        the rotation is around p when supplied otherwise
        it's the objects origin
          angle: angle in degrees, clockwise
          p: point to rotate around (external co-ords)
          return: reference to self

    def scale(self,sx,sy,p=None):
        scale object size (towards objects origin or p)
          sx sy: scale factors for each axis
          p: point around which to scale
          return: reference to self

    def itoe(self,p_i):
        convert internal to external co-ords
          p_i: intrnal co-ordinate
          return: external co-ordinate

    def etoi(self,p_e):
        convert external to internal co-ords
          p_e: external co-ordinate
          return: internal co-ordinate
```

A base class for objects that should implement affine transformations (such as scaling, rotating etc), this should apply to any object that draws on the page.

3.1.3. Area()

```
class Area(AffineObj):

    o=P(0,0)
    width=0
    height=0

    n, ne, e, se, s, sw, w, nw, c  ... see description below
```

A Rectangular area defined by the south-west corner and the width and height. This object mainly adds the ability to align to named compass points on the circumference see figure below.



These points are always returned in external co-ordinates.

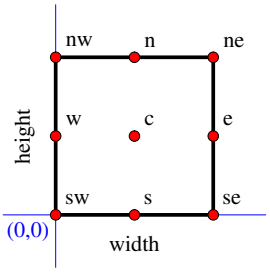
3.2.Drawing Objects

3.2.1. Common Attributes

Most of the objects that actually draw something on the page share a common set of attributes to set things like the line thickness etc.

- **fg**: A `Color()`, the colour for the ink in the foreground. Some objects allow switching this off with the value `None` in which case only the fill (if it's used) will be drawn.
- **bg**: A `Color()`, the fill color if the object supports this. A value of `None` means no fill (transparent).
- **linewidth**: The linewidth in pp.
- **linecap**: How to finish the ends of lines. 0=butt, 1=round, 2=square.
- **linejoin**: How to treat corners. 0=miter, 1=round, 2=bevel.
- **miterlimit**: Where to cut off the mitres (if you're using mitres in linejoins). 1.414 cuts off miters at angles less than 90 degrees, 2.0 cuts off miters at angles less than 60 degrees, 10.0 cuts off miters at angles less than 11 degrees, 1.0 cuts off miters at all angles, so that bevels are always produced.
- **dash:** The dash pattern to use for the forground lines. Currently this follows the postscript syntax. e.g. `"[]"` is a solid line, `"[2 3] 0 "` is a dashed line with ink for 2 pp gap for 3 pp and an initial offset for the ink of 0 pp. At some time in the future there may be a convenience class to set this.

3.2.2. Rectangle()

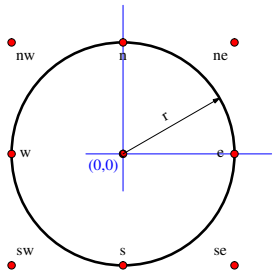


3.2.3. Circle()

```
bg=None
fg=Color(0)
r=1.0
```

```
start=0
end=360
linewidth=defaults.linewidth
dash=defaults.dash

n, ne, e, se, s, sw, w, nw, c  ... see description below

def locus(self,angle,target=None):
    Set or get a point on the locus

    @param angle: locus point in degrees
                  (Degrees clockwise from north)
    @param target: target point
    @return: target is None: point on circumference at that angle
             else: set point to the target, and return reference
                   to object
```

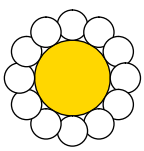
Draw a circle. The circle is specified by its position and its radius. You can also specify part of a circle with the attributes `start` and `end` which are in degrees clockwise from the top. As with the `Rectangle` there are named points on the enclosing square that corresponds to the compass points which can be read or set.



In addition an abtritrary point on the circumference can be read or set by using the `locus()` method — with one parameter (the angle on the locus) the locus point is returned; with an additional target point supplied, the locus point is set to the target point.

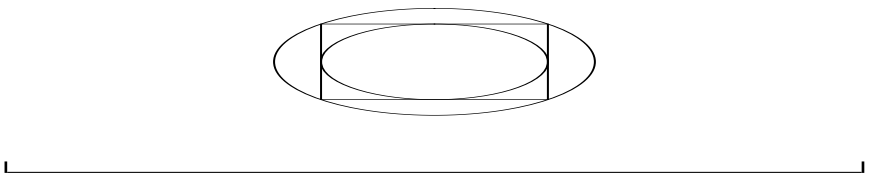
Example

```
c=Circle(r=.5,bg=Color('dandelion'))
g=Group()
for ii in range(0,360,30):
    g.append(
        Circle(r=.2,bg=Color('white')).locus(180+ii,c.locus(ii))
        )
render(c,g,file=...)
```



Example

```
g=Group(Rectangle(sw=P(0,0),width=2,height=2),
        Circle(r=1,sw=P(0,0)),
        Circle(r=sqrt(2)).locus(-135,P(0,0)),
        )
g.scale(1.5,.5)
render(g,file=...)
```



3.2.4. Dot()

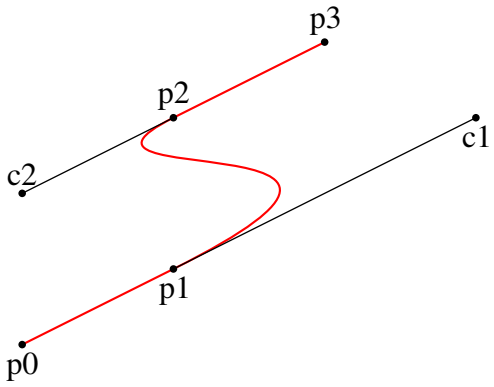
```
class Dot(Circle):
    r=.1
    bg=Color(0)
    fg=None
```

A simple convenience function to draw a dot at the given location

3.2.5. Path()

```
class Path(AffineObj):
    fg=Color(0)
    bg=None
    linewidth=None
    linecap=None
    linejoin=None
    miterlimit=None
    dash=None
    closed=0

        heads=[]

    length
    start
    end

    def P(self,t):
       point at fraction t of length

    def tangent(self,t):
       unit vector tangent to point at fraction t of length
```

An abritrary path (line curve etc). This is one of the more powerful objects in *PyScript*. `Path` takes a list of points (either `P` or `R`) in its contruction with a specifier for a curve (`C`) interdispersed if necessary. For example `Path(p0,p1,C(c1,c2),p2,p3)` would yield something like



The syntax is easy, between each pair of points, without a `C` inbetween, a straight line segment is drawn. If there is a `C` inbetween then a curve segment is drawn between the points. The curve is controlled by the arguments to `C`. At the moment only bezier curves can be specified by `C` which takes two arguments — they can be either a point or an angle. Points specify the control points of the bezier, and angles specify the direction the curve leaves the end points, the strength is calculated automatically.

The points to both the `Path` and `C` can be specified with either `P` or `R`. `P` are absolute points from the origin of the co-ordinate system, whereas `R` are 'relative' points and their meaning depends on where they appear. As arguments to `Path` then they are relative to the previous point (unless it is the first point in which case it the same as `P`). As arguments to `C` then for the first control point it's relative to the start of the curve segment and for the second control point it's relative to the end of the segment.

Example

The following two specify the same curve:

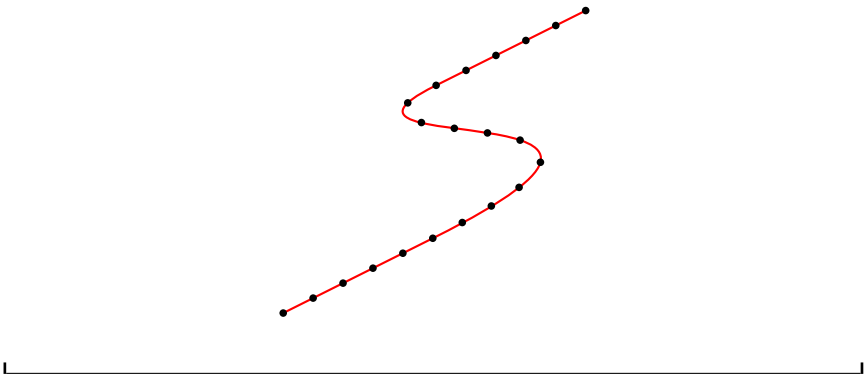
```
Path(P(1,1),P(2,2),C(P(2,3),P(3,2)),P(3,3))
Path(R(1,1),R(1,1),C(R(0,1),R(0,-1)),R(1,1))
```

As you would expect, you can set the linewidth, dash pattern, color etc in the usual way. If you set a `bg` that is not `None` then the 'interior' of the path will be filled in that color. The `heads` attribute is a list of `Arrowhead` *instances* to place on the path.

`Path` also has some additional methods and attributes. The `start` and `end` atributtes give, well, the start and end of the path. The `length` attribute give the length of the path in it's default co-ordinate system. You can also get the a point a fraction `f` along the path with `path.P(f)`, and a unit vector tangent to the point at that fraction with `path.tangent(f)`.

Example

```
path=Path(p0,p1,C(c1,c2),p2,p3,fg=Color('red'),linewidth=.8)

g=Group()
delta=1/20.
for p in range(21):
    g.append(Dot(path.P(p*delta)))
render(path,g,...)
```



3.2.6. Arrowhead()

```
class Arrowhead(AffineObj):
    fg=Color(0)
    bg=Color(0)

    reverse=0
    pos=1

    tip=P(0,0)
    angle=0

    start=(0,0)
    shape=[...]

    closed=1

    scalew=1
    scaleh=1

    linewidth=.2
    linejoin=2

    miterlimit = 2
```

A class to efficiently draw an arrow head. There are two ways of using the arrow head:

- Within the `Path` object: add arrowheads to the path by adding them to the `heads` attribute of `Path` which is a list. In this case, the arrow head takes one argument which is the fraction along the path at which to place the tip of the arrowhead. The arrow head can be reversed using its `reverse=1` attribute. eg `Path(...,heads=[Arrowhead(.5,reverse=1)])` will place an arrowhead halfway along the path pointing backward.

- Place the arrow head manually. You can set the tip position with `tip` and set the direction it points to with `angle`.

There are several styles of arrow heads defined `Arrowhead1` ... `Arrowhead5`, and new styles can be easily defined.

3.3. Text Objects

3.3.1. Text()

```
class Text(Area):
    A single line text object within an Area object

    text=''
    font="Times-Roman"
    size=12
    fg=Color(0)
    bg=None
    kerning=1
```

The *Text* object allows typesetting a simple string in a single font. The usual postscript fonts are defined, these are (case insensitive): *courier*, *courier_bold*, *courier_boldoblique*, *courier_oblique*, *helvetica*, *helvetica_bold*, *helvetica_boldoblique*, *helvetica_oblique*, *symbol*, *times_bold*, *times_bolditalic*, *times_italic*, *times_roman* and *zapfdingbats*.

The text will use *kerning* automatically, that is, the letter spacing will be adjusted depending on the pair of letters so that it looks nicer. The kerning can be turned of if necessary, see example below.

Example

```
t1=Text('SWEPT AWAY',kerning=0,size=20)
t2=Text('SWEPT AWAY',kerning=1,size=20,nw=t1.sw)
render(t1,t2,file=...)
```

Since *Text* is a subclass of *Area* then the usual compass points (*n*, *ne*, etc) are defined and can be read or set.

3.3.2. TeX()

```
class TeX(Area):
    an TeX expression

    text=""
    fg=Color(0)
```

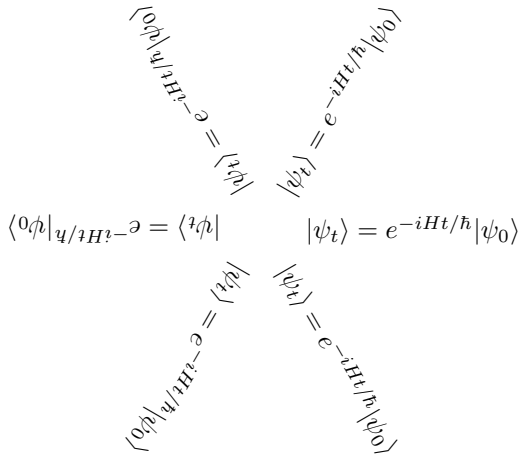
A LaTeX object — any LaTeXexpression, can be typeset and positioned on the diagram. The LaTeX expression is passed to the *latex* program followed by *dvips*, the resulting postscript is parsed and forms the basis of the object. Obviously this requires working *latex* and *dvips* distributions on your system. We recommend setting up your *latex* distribution to use postscript fonts, that way they can be scaled to any size.

One common pitfall is that the backslash ('\') is used in python strings as an escape character and so gets interpreted by python before the string gets passed to the *latex* program. The easiest work around to this problem is to use python raw-strings — just prepend an "r" to the string e.g. *r"α"*.

The object inherits from the *Area* object, and can also be scaled, rotated, etc. as will any of the other objects.

Example

```
tex=TeX(r'$|\psi_t\rangle=e^{itH}|\psi_0\rangle$',w=P(.5,0))

g=Group()
for ii in range(0,360,60):
    g.append(tex.copy().rotate(ii,P(0,0)))

render(g,file=...)
```



3.4. Groups

3.4.1. Group()

```
class Group(Area):

    def __init__(self,*objects,**dict):

    def append(self,*objs):
        append object(s) to group

    def apply(self,**dict):
        apply attributes to all objects

    def recalc_size(self):
        recalculate internal container size based on objects within

    def __getitem__(self,i):
    def __setitem__(self,i,other):
    def __getslice__(self,i,j):
    def __setslice__(self,i,j,wert):
```

This is one of the key classes in *PyScript*. `Group()` acts like a python list and groups together *PyScript* objects. Objects can be added to the group when you create it, e.g. `g=Group(det,b)`, or appended afterwards, e.g. `g.append(head,tail)`. You can access the items in the group as you would a normal python list, e.g. `head=g[2]`.

When an item is added to the group, the groups bounding box is recalculated and this allows the whole group to be positioned using `n`, `ne` etc. If you modify an object after it's been added to the group you will have to call the `.recalc_size()`— if you want the groups bounding box to reflect it's contents, you may not want this under certain applications.

Groups()'s can be nested without problem. All the items will be rendered in the order they where added.

The properties of the groups contents can be set *en-masse* by using the *.apply()* method. Objects that don't understand a particular property will be skipped. e.g. *g.apply(linewidth=2)*.

3.5. Vectors and Matrices

3.6. Other

3.6.1. Color()

```
class Color(PsObj)
    def __mul__(self,other)
```

This class represents a postscript color. There are four ways to specify the color distinguished by the number and type of paprameters that are passed when you create the object.

- *Color(C,M,Y,K)* - a postscript CMYKColor (Cyan, Magenta, Yellow, blacK)
- *Color(R,G,B)* - RGBColor (Red, Green, Blue)
- *Color(G)* - Gray
- *Color('Yellow')* etc

All the numbers above range from 0 to 1. Some of the named colors that are defined are Red, Green, Blue, Cyan, Magenta, Yellow, Black, White.

Color objects can be multiplied by a numeric factor. The effect is mostly to darken colors if the factor is less than 1 and to lighten colors if it's greater, but this depends on how the colors where specified. eg *Color(.2,.6,.6)*.5 = Color(.1,.3,.3)*

The colours in the named colour model are shown in figure 3.1. As a historical note, the color names originated from unixes X11 color names, and were at one point considered as cadidate named colours for HTML documents, but in the end where never adopted. They have however, aquired an unofficial permanence.

A final note on the colors — what you get on paper may not reflect what you see on the screen. The actual color that turns up on the paper is a complicated function of how it was produced, and depends on the hardware. The fastest and most accurate way to match colors in a printed document is to print out a color chart on the intended hardware.

3.6.2. Paper()

```
class Paper(Area):
    PAPERSIZES={"a0", ...''letter'', ...}
```

This is a convenience class, just an *Area()* with predefined size given by the usual paper sizes such as "a4", "letter" and "legal" etc. The origin is at the sw corner. It's useful if you want an object that will help align things on a printed page. e.g. *page=Paper("a4")*.

3.6.3. Epsf()

```
class Epsf(Area):
```

Figure 3.1: Named colors

Include an encapsulated postscript file (eps) in the figure. An eps file is a single page postcript file describing a diagram. There are many programs, such as graphing programs, that will generate eps files as output. It has to obey certain rules, such as having no page brakes, and a bounding box. *PyScript* will parse the file and extract the bounding box and use that as the basis of the size and placement of the figure (so if it's wrong don't blame *PyScript*). *Epsf()* takes a single argument — the path of the eps file. The resulting object can then be positioned using `n`,`c`,`ne` etc, and of course can be scaled and rotated as desired.

The Eps file can also be scaled to a particular width or height by specifing either the *width* or the *height* attributes when you create the object. The aspect ratio will be preserved when you do this. If you give *both* width and height attributes the object will be scaled to those dimensions without preserving its aspect ratio.

4: Development

The aim of this section is to document some of the internals of *PyScript* to enable developers to modify and extend it. It should also help in solving some of the trickier problems.

4.1. **Submitting patches**

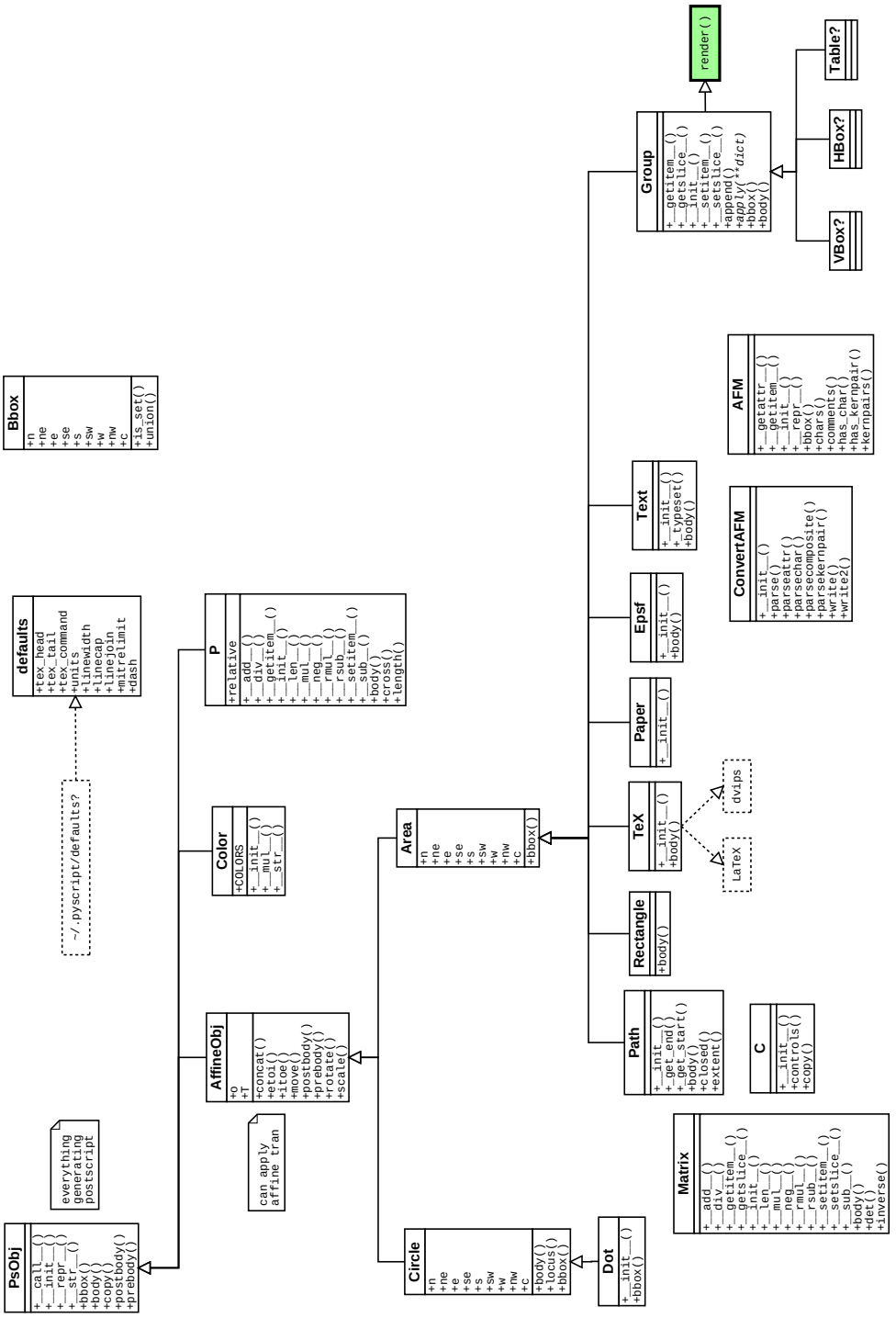


Figure 4.1: Class structure of *PyScript*

A: Presentation Library

```
pyscript.lib.present
```

It's straightforward to create whole posters in *PyScript*. Talks can also be simply created.

A.1.Common Objects

These are useful objects for both posters and talks.

A.1.1. Box()

```
class Box(Group, Rectangle):
    pad = .2

    width = None
    height = None
```

This places a box around an object. *pad* gives the amount of padding around the object. *width* and *height* can be set also and these will override the calculated values.

A.1.2. TeXArea()

```
class  TeXArea(Group):
    width = 9.4
    iscale = 1
    align = "w"
```

This will typeset some LaTeX within a fixed width minipage environment. the width of the minipage is set with the *width* attribute which must be supplied. The initial scale of the text can be set with *iscale* as per the *TeX* object. The *align* attribute is used when the LaTeX doesn't fill the minipage — an invisible rectangle is added with the minipage aligned according to this attribute.

A.2.Posters

```
class Poster(Page, VAlign):
    size = "A0"
```

```
    orientation = "portrait"

    bg = Color('DarkSlateBlue')

    space = 1

    topspace = 2

    def background(self)
```

A.3. Talks

```
class Pause(object)
class Talk(Pages):
    def append(self, *slides_raw):
class EmptySlide(Page):

    title = None
    orientation = "Landscape"
    size = "screen"

    def flatten(self, thegroup=None, objects=[]):
    def append(self, *items, **options):
    def append_n(self, *items):
    def append_s(self, *items):
    def append_e(self, *items):
    def append_w(self, *items):
    def append_c(self, *items):

    def make_back(self):
    def make_title(self):

        def clear(self):
    def make(self, page=1, total=1):
```

B: The Old Presentation Library

`pyscript.lib.presentation`

In addition to the newer `pyscript.lib.present` library, there also exists the old `pyscript.lib.presentation` library. This is not quite so object-oriented in usage, but it works nevertheless, and hasn't been completely deprecated. The `presentation` library can be used to create posters and talks, which can then be used to "wow" your colleauges at your next conference.

B.1.Common Objects

These are useful objects for both posters and talks.

B.1.1. TeXBox()

Typeset some LaTeX within a fixed width box.

B.1.2. Box_1()

A box of fixed width. Items added to it are aligned vertically and separated by a specified padding.

B.2.Creating a talk or seminar

B.2.1. The Talk() object

The first thing you will need to do when you start writing a seminar presentation is to instantiate the `Talk()` object. This object defines some overall parameters, attributes and styles for the talk as a whole. After you set these parameters for your particular talk, then you only need to worry about adding `Slide()` objects. To set up the `Talk()` object for your talk you merely need to do this:

```
talk = Talk()
```

More interesting things happen when we add styles to the talk, but more on that later. If you want to know about that now, go to Section B.2.3.

The next thing you probably want to do is to give your talk a title. There are two ways to do this: with the *set_title()* method, or by merely setting the *title* attribute of the instantiated *Talk()* object. In other words you can either do this:

```
talk.set_title(r"This is my talk")
```

or this:

```
talk.title = r"This is my talk"
```

It is common that there are many people who have contributed to a particular piece of work being discussed in the seminar or talk, and consequently there will be more than one "author" of the talk. However, there is usually only one person presenting the talk, and so we have two separate attributes for these situations, namely the *authors* and *speaker* attributes. To set the name of the authors contributing to the talk, either use the *set_authors()* method, or set the attribute directly, like so:

```
talk.set_authors(r"Tom, Dick, and Harry")
```

or:

```
talk.authors = r"Tom, Dick and Harry"
```

For the speaker, this is almost an identical procedure, just use either the *set_speaker()* method, or set the *speaker* attribute directly.

You are likely to be representing a business or institute of some form, so it is best to give their address. To do provide this information to the *presentation* library so that it can place the text appropriately, just use the *set_address()* method or set the *address* attribute of the instantiated *Talk()* object.

It is possible that your business or institute has a logo that you'd like to use. If so, convert it to an EPS file, and you can add it to the talk using the *add_logo()* method like so:

```
talk.add_logo("myepslogo.eps", height=2)
```

You can set the height of the logo using the *height* attribute as shown in the example above.

That's basically it as far as the `Talk()` object itself goes. The main amount of work is in producing the individual slides of the presentation, which is what we discuss next.

B.2.2. The Slide() object

The *Slide()* object defines a particular slide; one creates a new *Slide()* object for each slide in the presentation, calling them all at the end in the *render()* function to generate the entire talk.

The first slide in your talk will be the titlepage. However, since getting you to make a new *Slide()* object just to generate the title page is silly (well, you've just given the title page all it needs to know eh?) the library automatically generates the title page slide for you.

Slides usually have a title, some sequence of headings, possibly a figure (defined in *PyScript* for example), or an imported EPS image, and possibly some equations. The *presentation* library provides methods for doing all these things. To add a slide to the presentation, one must instatiate a new *Slide* object, passing to it the current *Talk* object, like so:

```
intro = Slide(talk)
```

To add a title to the slide [1], one can use the *set_title()* method (this time of the *Slide* class), or set the *title* attribute directly:

[1] Note that this is **not** the title of the talk!

```
intro.set_title(r"Introduction")
```

or:

```
intro.title = r"Introduction"
```

There isn't much of a difference between the two I know, but some people like to call a `set_` function and others like to set the attribute directly, so we're catering to both kinds of people.

To add other things like headings, figures and epsf images to your slide, you just need to use one of the relevant `add_*()` functions. In other words, to add a heading use the `add_heading()` method. This method takes two arguments, the first argument is the level of the heading (there are currently three separately defined levels of headings in the library) and the second argument is the heading to add. For instance:

```
intro.add_heading(1, r"What are we talking about?")
intro.add_heading(2, r"Some stuff")
intro.add_heading(2, r"Some other stuff")
intro.add_heading(3, r"Something more specific to some other stuff")
```

Adding a heading puts one of the predefined bullets in front of the heading, typesets the text at a predefined size and indentation dependent upon the heading level. You can change these settings by defining your own style, or by setting one of the myriad attributes of the talk object itself, for more information see Section B.2.3.

If you want to place a diagram generated from *PyScript* code to your slide, and have it automatically positioned by the library, then use the `add_fig()` method. For instance, if you've produced earlier in your *PyScript* script a diagram called `mydiag` then to add it to the slide, merely use:

```
intro.add_fig(mydiag)
```

You can set the location of the figure by passing one of the *PyScript* anchor locations as an optional argument. For example,

```
intro.add_fig(mydiag, ne=intro.area.ne - P(1,1))
```

will locate the diagram in the "north-east" corner of the page one centimetre from the right-hand edge, and one centimetre from the top edge. You can set the width of the diagram as well by specifying the `width` option:

```
intro.add_fig(mydiag, width=12, c=intro.area.c)
```

which will make the diagram 12cm wide, and centre it on the current page.

Similarly, one can add diagrams or images that already exist in EPS files. To do this use the `add_epsf()` method like so:

```
intro.add_epsf(file="myEpsFile.eps", c=intro.area.c, width=14)
```

The `add_epsf()` method processes the anchor location, width and height options in the same way the `add_fig()` method does.

You might like to add some text to your slide and position it arbitrarily on the page, as opposed to have `presentation.py` work out where to put it for you (as is done with the `add_heading()` method). Therefore, there is the convenience function `add_text()`. Here is an example:

```
intro.add_text("Hello there!", e=intro.area.e-P(-2,-2), scale=2)
```

Just to be different, the *add_text()* method has a scale attribute as opposed to a height or width attribute. This is really silly, and should probably be changed. If you read this sentence, please put a feature request to change this on the pyscript web page :-).

At the end of your script you'll want to make the talk in its entirety, to do this use the *make()* method of the *Talk()* class like so:

```
# make it!
talk.make(
    intro,
    another_slide,
    file="mytalk.ps")
```

This will generate a Postscript document called *mytalk.ps* in the same directory as that in which the *PyScript* script was run. Note that this is a Postscript file and not an EPS file; this means that the output has multiple pages as one would hope for a seminar!

To actually give your seminar there are several tools you can use. One of the most common is to turn the Postscript into PDF via `ps2pdf` or some similar tool, and then use the full screen mode of Adobe Acrobat Reader [2] to display the talk. Alternatively, you might like to use a program like `pspresent` [1]. Please note that due to a bug in Ghostscript version 7 (and possibly below) if there is insufficient text on a page, the postscript will be converted by `ps2pdf` incorrectly to give a portait-orientated page, as opposed to a landscape page. The postscript will view correctly in Ghostview (and so will the pdf incidentally), however, the pdf will view incorrectly in `xpdf` and `acroread`. This bug seems to have been fixed in Ghostscript version 8.

B.2.3. Styles for talks and seminars

To change your talk style from the default you can specify one of the predefined styles by passing the *style* option to the *Talk()* class on instantiation. For instance:

```
talk = Talk(style="prosper-darkblue")
```

which will load the style that looks a lot like the "darkblue" style of the prosper LaTeX package. To load the style, *PyScript* will look in either the `/.pyscript/styles/` directory or the current directory for a python file whose file name will be the name of the style with '`.py`' appended.

If you are feeling really keen, you can write your own style. One of the best ways to do this is to copy and then modify one of the ones provided with the *PyScript* distribution. Let's go through the details of the "prosper-darkblue" style now.

```
# talk style for PyScript, following the Darkblue design of prosper

HOME = os.path.expandvars("$HOME")
stylesDir = HOME + "/.pyscript/styles/"

# set the foreground and background colour of the title text of the talk
self.title_fg = Color('white')
self.title_bg = Color('white')

# set the talk title's text style
self.title_textstyle = r"\bf\sf"

# set the text style for the text of who is giving the talk
self.speaker_textstyle = r"\sf"

# set the colour and text style of the address of the speaker of the talk
self.address_fg = Color('white')
self.address_textstyle = r"\sf"

# set the colour and text style of the authors of the talk (not necessarily
```

```
# the speaker of the talk)
self.authors_fg = Color('white')
self.authors_textstyle = r"\sf"

# set the colour and text style of the title of the *slide*
self.slide_title_fg = Color('lightgray')
self.slide_title_textstyle = r"\bf"

# set the colour, scale, textstyle, bullet and indent type for a level 1 heading
self.headings_fgs[1] = Color('white')
self.headings_scales[1] = 3
self.headings_textstyle[1] = r"\sf"
self.headings_bullets[1] = Epsf(file=stylesDir+"redbullet.eps").scale(0.2,0.2)
self.headings_indent[1] = 0

# set the colour, scale, textstyle, bullet and indent type for a level 2 heading
self.headings_fgs[2] = Color('white')
self.headings_scales[1] = 2.5
self.headings_textstyle[2] = r"\sf"
self.headings_bullets[2] = Epsf(file=stylesDir+"greenbullet.eps").scale(0.15,0.15)
self.headings_indent[2] = 0.5

# set the colour, scale, textstyle, bullet and indent type for a level 3 heading
self.headings_fgs[3] = Color('white')
self.headings_scales[1] = 2.2
self.headings_textstyle[3] = r"\sf"
self.headings_bullets[3] = Epsf(file=stylesDir+"yellowbullet.eps").scale(0.1,0.1)
self.headings_indent[3] = 1

# set the colour, textstyle and scale for placed text
self.text_scale = 3.0
self.text_fg = Color('white')
self.text_textstyle = r"\sf"
```

First off we work out where the styles directory is, and since it should be in `/.pyscript/styles` we define this with the `stylesDir` variable. Note that this is a python file like any other, all it's doing is expecting you to add a certain set of values that *PyScript*'s *presentation.py* library knows about.

Next we set the title foreground and background colour by setting the *title_fg* and *title_bg* variables. The text style is bold (
`bf`) and a sans serif (
`sf`) font. Note that this uses LaTeX to specify these styles as internally *PyScript TeX* objects are used. If this is a problem for you, just send us a bug report or feature request on the pyscript web page and we'll try and fix it for you as soon as we can.

Next the text style for the speaker element of the title page is set, and the colour and text style for the authors and address elements.

Now we process the elements for the slides themselves. We need to set a foreground colour and text style for the title of the slide. We also need to specify the scale, foreground colour, text style bullet and indentation amount to use for the three levels of headings defined in the *presentation.py* library. Note that one has to specify for which level the style is being set by the square brackets: e.g. *headings_fg[1]*.

One doesn't have to set the bullets to a text or LaTeXkind of string, but also to EPS files, which is what we have done in the above example where red, green and yellow circles are used for the various heading levels. We've scaled them here (using the *PyScript scale* method) so that they have different sizes and one can more easily tell when viewing the talk that one level has precedence over the other.

If one wishes to place arbitrary text on the page, one can specify the default scale, colour and text style here too.

Styles can be found in the *contrib/* directory of the *PyScript* distribution.

B.3. Creating a poster

B.3.1. The Poster() object

The very first thing to do when making a new poster is to instantiate the `Poster()` object. You do this like so:

```
poster = Poster(size="a4")
```

Note that the size of the poster has been set here explicitly. A handy thing about using postscript is that if you want an a0 poster, you just need to change the size paramter to "a0" and the size of the poster will change, but not the actual amount of postcript. The reason we add this size declaration to the poster class is because it is often handy to have a4 size versions of your poster when you are at a poster session at a conference to give to people who have viewed your a0 size poster.

Your poster will then require a title, a list of authors, an address of the insititution you are representing and the abstract that you submitted (or are using) for the poster. You set these properties using the relevant `set_()` methods of the `Poster()` class. For instance:

```
poster.set_title("My poster")

poster.set_authors("Me, him, and her")

poster.set_address("Over the hills, and far away")

poster.set_abstract("""
I should have written something better when I applied for the conference: I
might have got a talk instead...
""")
```

One often wants to put logos onto a poster. The `Poster()` class lets you do this easily by using the `add_logo()` method. Just supply the name of an EPS file and the height you want to use for the logo, and the class will place it at the top of your poster for you. E.g.:

```
poster.add_logo("my_first_logo.eps", height=1.2)
poster.add_logo("my_second_logo.eps", height=1.2)
```

The first logo is placed at the top left-hand corner. The next logo will be placed in the top right-hand corner. If you specify more logos then they will be distributed evenly across the top of the poster. If you want, you can also specify a list of logo names all at the one time, instead of having to call `add_logo()` several times. You do this with the `add_logos()` method:

```
poster.add_logos("my_first_logo.eps", "my_second_logo.eps", height=1.2)
```

Obviously, when you use this method, all logos are given the same height.

A poster is usually split up into columns. For a portrait-orientated poster, one usually has two columns; for a landscape-orientated poster three columns. A column is made up of one or more "column boxes" which define the actual content in the poster. So, to make your first column, you need to make a new `Column()` object:

```
col1 = Column(poster)
```

Note that we have passed the current `poster` object through as an argument. This is so that any styles defined in the poster can flow through to the subobjects. We will eventually add this column to the poster as a whole, but not yet, as we need to fill the column with some content first. To do this instantiate a `ColumnBox()` object like so:

```
intro = ColumnBox(poster)
intro.set_title("Introduction")
intro.add_TeXBox(r"""
Here is some \LaTeX{}
""")
```

Again, the `poster` object is passed in as an argument so we can make use of previously defined styles. We then set the title of the column box with the `set_title()` method, and then added a `TeXBox()` object, which can contain arbitrary LaTeX expressions, with the `add_TeXBox()` method.

There are more things than just `TeXBox`es that you can add to a column box; you can add an already-defined *PyScript* object, or an EPS file. Say you want to align two images up together, and display them side by side within the column box. To do this you would set up an `Align()` object, append the EPS files as `Epsf()` objects and then add the `Align()` object to the column box with the `add_object()` method. E.g.:

```
# the figures
fig1 = Epsf("fig1.eps")
fig2 = Epsf("fig2.eps")

# make the Align object
figs = Align(a1="ne", a2="nw", angle=90, space=0.2)
figs.append(fig1, fig2)

# add the Align object to the column box
intro.add_object(figs)
```

The `add_epsf()` method is merely a convenience function so that you don't have to define an `Epsf()` object first and then add it with the `add_object()` method. You can do all that work if you want to though!

We are now ready to add the column box to the column. We do this with the `add_box()` method of the `Column()` object:

```
col1.add_box(intro)
```

We can now add the column to the poster itself, and yes, you guessed it, to do this we use the `add_column()` method of the `Poster()` object:

```
poster.add_column(col1)
```

Since this is the first column, it will automatically be the left-most column. Any other columns that you add will be added to the right of it. So, for instance, if one were making a poster in portrait mode then one would simply add two columns; the `Poster()` class should handle aligning most of the bits and pieces for you.

For more complex examples, have a look in the `examples/` directory of the *PyScript* distribution. The `Poster()` class is a complete rewrite and extension of the older `Poster_1()` class, so it is likely that there will be problems with the way that it has been designed and so the interface might change slightly in the future. If you are having problems, feel free to email the developers! If you read this sentence, you might want to email the developers anyway! :-)

B.3.2. Styles for posters

To change your poster style from the default you can specify one of the predefined styles by passing the `style` option to the `Poster()` class on instantiation. For instance:

```
poster = Poster(size="a4", style="ccp2004-poster")
```

which will load the style that PTC used at the Conference on Computational Physics in 2004 (hence ccp2004). Just like with styles for talks, to load the style, *PyScript* will look in either

the `/.pyscript/styles/` directory or the current directory for a python file whose file name will be the name of the style with '`.py`' appended.

If you are feeling really keen, you can write your own style. One of the best ways to do this is to copy and then modify one of the ones provided with the *`PyScript`* distribution.

Styles can be found in the *`contrib/`* directory of the *`PyScript`* distribution.

C: Quantum Information Library

D: PyScript Optics Object Package

This package is a library of functions and objects for use in constructing optical circuits such as those used in interferometers and optical setups useful in scientific applications.

D.1. Examples

D.1.1. Michelson-Morely Interferometer

This example shows how to construct one of the simplest interferometers with *PyScript*. A laser is incident onto a 50:50 beam splitter, the light beam then being split equally into the two "arms" of the interferometer. At the end of each arm is a mirror which reflects the light directly back to the beam splitter which recombines the light, and the output of the interferometer is measured at the detector (below the beam splitter in the diagram). As the length of one of the arms changes the voltage measured at the detector will vary sinusoidally. In this example we are using the `Laser`, `Mirror`, `BSBox` and `Detector` objects of the `optics` library.

```
# Michelson-Morely interferometer

# import the pyscript objects
from pyscript import *
# import the optics library
from pyscript.lib.optics import *

# set up some handy defaults
defaults.units=UNITS['cm']

# initialise a laser beam
beam = Group()

# the laser
laser = Laser(c=P(0,0))

# the beam splitter
bs = BSBox(height=0.7)
bs.w = laser.e + P(1,0)
beam.append(Path(laser.e, bs.w))

# the "north" mirror
mirror_n = Mirror(angle=90)
mirror_n.s = bs.n + P(0,3)
beam.append(Path(bs.n, mirror_n.s))
```

```
# the "east" mirror
mirror_e = Mirror()
mirror_e.w = bs.e + P(3,0)
beam.append(Path(bs.e, mirror_e.w))

# the detector
det = Detector(angle=90)
det.n = bs.s + P(0,-1)
beam.append(Path(bs.s, det.n))

# make the beam red
beam.apply(fg=Color("red"))

# collect all the objects together
fig = Group(
        laser,
        bs,
        mirror_n, mirror_e,
        det,
        beam,
        )

# render the figure
render(fig,
        file="michelson-morely.eps")
```

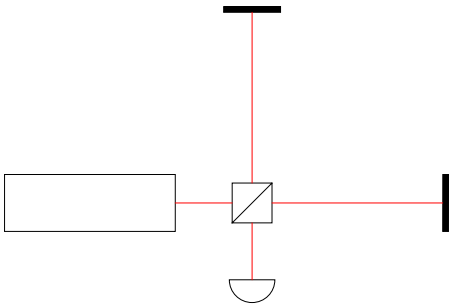


Figure D.1: Michelson-Morely interferometer

D.1.2. Mach-Zehnder Interferometer

Another commonly used interferometer is the Mach-Zehnder interferometer. In this design there are two beam splitters. The light is split on the first beam splitter, and travels down each arm of the interferometer to two mirrors which recombine the light on the second beam splitter, the output being detected by two detectors at the two output ports of the second beam splitter. This interferometer is commonly used for explaining such concepts as Quantum Non-Demolition experiments, and other "quantum weirdness" associated with light.

```
# Mach-Zehnder interferometer

# import the pyscript objects
from pyscript import *
# import the optics library
from pyscript.lib.optics import *

# set up some handy defaults
defaults.units=UNITS['cm']

# initialise a laser beam
beam = Group()

# the laser
```

```
laser = Laser(c=P(0,0))

# the "west" beam splitter
bs_w = BSBox(height=0.7)
bs_w.w = laser.e + P(1,0)
beam.append(Path(laser.e, bs_w.w))

# the "north" mirror
mirror_n = Mirror(angle=45)
mirror_n.s = bs_w.n + P(0,3)
beam.append(Path(bs_w.n, mirror_n.c))

# the "east" mirror
mirror_e = Mirror(angle=45)
mirror_e.w = bs_w.e + P(3,0)
beam.append(Path(bs_w.e, mirror_e.c))

# the "east" beam splitter
bs_e = BSBox(height=0.7)
bs_e.c = P(mirror_e.c.x, mirror_n.c.y)
beam.append(Path(mirror_e.c, bs_e.s))
beam.append(Path(mirror_n.c, bs_e.w))

# the "north" detector
det_n = Detector(angle=-90)
det_n.s = bs_e.n + P(0,1)
beam.append(Path(bs_e.n, det_n.s))

# the "east" detector
det_e = Detector()
det_e.w = bs_e.e + P(1,0)
beam.append(Path(bs_e.e, det_e.w))

# set the colour of the beam
beam.apply(fg=Color("red"))

# collect all the objects together
fig = Group(
        laser,
        bs_w,
        mirror_n, mirror_e,
        bs_e,
        det_n, det_e,
        beam,
        )

# render the figure
render(fig,
        file="mach-zehnder.eps")
```

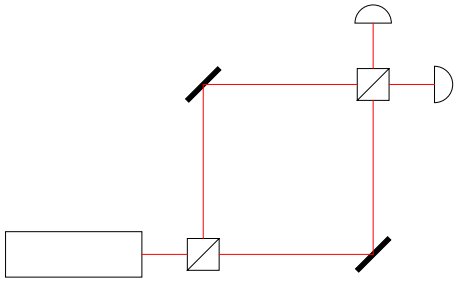


Figure D.2: Mach-Zehnder interferometer

D.1.3. Sagnac Interferometer

The Sagnac interferometer is another of the major interferometer configurations. The light is split on the beam splitter and procedes around the interferometer in opposite directions then being recombined on the same beam splitter and the output interference is read at the output port (where the detector is in the diagram This interferometer can be used in optical gyroscopes because a rotation in the plane of the interferometer can be measured since the light traveling in one direction travels a different distance to the light traveling in the other direction, hence giving a path difference and interference fringes. An interesting point about the script below is that the "north-east" mirror has its location set by the y position of the centre of the "north" mirror and its x position set by the centre of the "east" mirror. This means that if one changes the location of either of these two mirrors then the position of the "north-east" mirror changes appropriately.

```
# Sagnac interferometer

# import the pyscript objects
from pyscript import *
# import the optics library
from pyscript.lib.optics import *

# set up some handy defaults
defaults.units=UNITS['cm']

# initialise a laser beam
beam = Group()

# the laser
laser = Laser(c=P(0,0))

# the beam splitter
bs = BSBox(height=0.7)
bs.w = laser.e + P(1,0)
beam.append(Path(laser.e, bs.w))

# the "north" mirror
mirror_n = Mirror(angle=45)
mirror_n.s = bs.n + P(0,2)
beam.append(Path(bs.n, mirror_n.c))

# the "east" mirror
mirror_e = Mirror(angle=45)
mirror_e.w = bs.e + P(3,0)
beam.append(Path(bs.e, mirror_e.c))

# the "north-east" mirror
mirror_ne = Mirror(angle=135)
mirror_ne.c = P(mirror_e.c.x, mirror_n.c.y)
beam.append(Path(mirror_n.c, mirror_ne.c))
beam.append(Path(mirror_e.c, mirror_ne.c))

# the detector
det = Detector(angle=90)
det.n = bs.s + P(0,-1)
beam.append(Path(bs.s, det.n))

# set the colour of the beam
beam.apply(fg=Color("red"))

# collect all the objects together
fig = Group(
        beam,
        laser,
        bs,
        mirror_n, mirror_e, mirror_ne,
```

```
        det,
        )

# render the figure
render(fig,
        file="sagnac.eps")
```

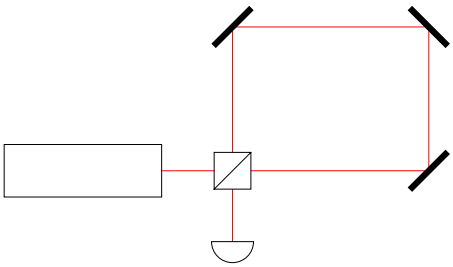


Figure D.3: Sagnac interferometer

D.1.4. A Fabry-Perot Cavity

A Fabry-Perot cavity is basically just two parallel mirrors facing one another and a laser field injected into the cavity via one of the mirrors (which might be 95% reflective, say). In the setup shown here we have a Pound-Drever-Hall setup which can allow for stabilisation of the cavity via the electro-optical modulator (EOM). The diagram in Figure D.4 shows such a cavity with a `FreeSpace()` object used as well. This is something important for gravitational wave detectors as they have a lot of free space in the arms of the interferometers! Also introduced in this example is the `Modulator()` object.

```
# a Fabry-Perot cavity in a Pound-Drever-Hall setup

# import the pyscript objects
from pyscript import *
# import the optics library
from pyscript.lib.optics import *

# set up some handy defaults
defaults.units=UNITS['cm']

# initialise a laser beam
beam = Group()

# the laser
laser = Laser(c=P(0,0))

# the EOM
eom = Modulator()
eom.w = laser.e + P(1,0)
beam.append(Path(laser.e, eom.w))

# the "west" mirror
mirror_w = Mirror()
mirror_w.w = eom.e + P(1,0)
beam.append(Path(eom.e, mirror_w.w))

# some free space
fs = FreeSpace()
fs.w = mirror_w.e + P(1,0)
beam.append(Path(mirror_w.e, fs.w))

# the "east" mirror
mirror_e = Mirror()
```

```
mirror_e.w = fs.e + P(1,0)
beam.append(Path(fs.e, mirror_e.w))

# set the colour of the beam
beam.apply(fg=Color("red"))

# collect all the objects together
fig = Group(
        beam,
        laser,
        eom,
        mirror_e, mirror_w,
        fs,
        )

# render the figure
render(fig,
        file="fabry-perot_pdh.eps")
```

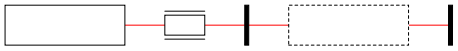


Figure D.4: Fabry-Perot cavity in Pound-Drever-Hall setup.

D.2. Objects

D.2.1. BSBox

A beam splitter shown as a box. Some beam splitters are two prisms of glass bonded together and look like a box with a line along the diagonal when viewed from above, hence the look of this object.

Object options:

height: The height of the beam splitter, its width is equal to its height.

angle: Rotation angle. The beam splitter can be returned already turned to the desired angle (in degrees).

fg: The foreground colour. Use the `Color` object to set this option.

bg: The background colour. Use the `Color` object to set this option.

Figure D.5: BSBox object

D.2.2. BSLine

A beam splitter shown as a thin box at a default angle of 45 degrees. Some beam splitters are partially silvered mirrors, hence the look of this object.

Object options:

height: The height of the beam splitter.

thickness: The thickness of the beam splitter.

angle: Rotation angle. The beam splitter can be returned already turned to the desired angle (in degrees). The default angle is 45 degrees.

fg: The foreground colour. Use the `Color` object to set this option.

bg: The background colour. Use the `Color` object to set this option.

Figure D.6: BSLine object

D.2.3. Detector

A simple D-shaped detector symbol.

Object options:

height: The height of the detector

width: The width of the detector.

angle: Rotation angle. The detector can be returned already turned to the desired angle (in degrees).

pad: Space padding around the object.

fg: The foreground colour. Use the `Color` object to set this option.

bg: The background colour. Use the `Color` object to set this option.

Figure D.7: Detector object

D.2.4. Free Space

This object generates a dashed box where the free space is. This can be useful if one wants to highlight that the free space in one arm of an interferometer has a particular refractive index, or some other property.

Object options:

height: The height of the free space region.

width: The width of the free space region.

angle: Rotation angle.

fg: The foreground colour. Use the `Color` object to set this option.

bg: The background colour. Use the `Color` object to set this option.

Figure D.8: Free Space object

D.2.5. Lambda Plate

This object is used to describe changes in an optical beam of, for example, a half or a quarter of a wavelength.

Object options:

height: The height of the lambda plate.

width: The width of the lambda plate.

angle: Rotation angle.

fg: The foreground colour. Use the `Color` object to set this option.

bg: The background colour. Use the `Color` object to set this option.

Figure D.9: Lambda Plate object

D.2.6. Laser

At present all this object does is generate a simple box. However, in the future we hope to make this somewhat better looking.

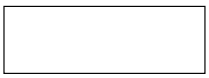


Figure D.10: Laser object

D.2.7. Lens

Generate a convex or concave lens.

Object options:

height: The height of the lens.

thickness: The thickness of the lens.

angle: Rotation angle.

type: A string specifying if the lens is convex or concave. Default is `concave`.

fg: The foreground colour. Use the `Color` object to set this option.

bg: The background colour. Use the `Color` object to set this option.

Figure D.11: Lens object

D.2.8. Mirror

A very simple mirror symbol. At present doesn't handle concave/convex mirrors, but will do hopefully in the future.

Object options:

length: The length of the mirror.

thickness: The thickness of the mirror.

angle: Rotation angle.

flicks: Should "flicks" indicating the back of the mirror be put on? This is a boolean value, and by default this is `False`.

fg: The foreground colour. Use the `Color` object to set this option.

bg: The background colour. Use the `Color` object to set this option.

Figure D.12: Mirror object

D.2.9. Modulator

Generates a modulator symbol, such as for an acousto-optical modulator, or electro-optical modulator. This is simply a box with two lines on either side.

Object options:

height: The height of the modulator.

width: The width of the modulator.

angle: Rotation angle.

fg: The foreground colour. Use the `Color` object to set this option.

bg: The background colour. Use the `Color` object to set this option.

Figure D.13: Modulator object

D.2.10. Phase Shifter

Produces a triangle shape with the point directed upwards by default. This component is used in optics to vary the phase of an optical signal.

Object options:

height: The height of the phaser shifter.

width: The width of the phaser shifter.

angle: Rotation angle.

fg: The foreground colour. Use the `Color` object to set this option.

bg: The background colour. Use the `Color` object to set this option.

Figure D.14: Phase Shifter object

E: PyScript Electronics Object Package

E.1.**Introduction**

This package is a library of standard symbols as used in electronic circuit layout and design. Thanks to Adrian Jonstone's lcircuit macros from CTAN for the ideas and names.

E.2.**Objects**

E.2.1. AND gate

Draws a standard AND gate, with pins extending from the body of the gate.

Object options:

width: The width of the object.

height: The height of the object.

angle: The rotation angle.

pinLength: The length of the pins extending from the gate.

fg: The foreground colour. Use the `Color` object to set this option.

bg: The background colour. Use the `Color` object to set this option.

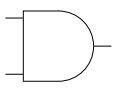


Figure E.1: AndGate object

E.2.2. NAND gate

Draws a standard NAND gate, with pins extending from the body of the gate.

Object options:

width: The width of the object.

height: The height of the object.

angle: The rotation angle.

pinLength: The length of the pins extending from the gate.

fg: The foreground colour. Use the `Color` object to set this option.

bg: The background colour. Use the `Color` object to set this option.

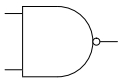


Figure E.2: NandGate object

E.2.3. OR gate

Draws a standard OR gate, with pins extending from the body of the gate.
Object options:

width: The width of the object.

height: The height of the object.

angle: The rotation angle.

pinLength: The length of the pins extending from the gate.

fg: The foreground colour. Use the `Color` object to set this option.

bg: The background colour. Use the `Color` object to set this option.

Figure E.3: OrGate object

E.2.4. NOR gate

Draws a standard NOR gate, with pins extending from the body of the gate.
Object options:

width: The width of the object.

height: The height of the object.

angle: The rotation angle.

pinLength: The length of the pins extending from the gate.

fg: The foreground colour. Use the `Color` object to set this option.

bg: The background colour. Use the `Color` object to set this option.

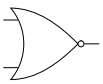


Figure E.4: NorGate object

E.2.5. XOR gate

Draws a standard XOR gate, with pins extending from the body of the gate.

Object options:

width: The width of the object.

height: The height of the object.

angle: The rotation angle.

pinLength: The length of the pins extending from the gate.

fg: The foreground colour. Use the `Color` object to set this option.

bg: The background colour. Use the `Color` object to set this option.

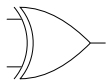


Figure E.5: XorGate object

E.2.6. NXOR gate

Draws a standard NXOR gate, with pins extending from the body of the gate.

Object options:

width: The width of the object.

height: The height of the object.

angle: The rotation angle.

pinLength: The length of the pins extending from the gate.

fg: The foreground colour. Use the `Color` object to set this option.

bg: The background colour. Use the `Color` object to set this option.

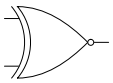


Figure E.6: NxorGate object

E.2.7. NOT gate

Draws a standard NOT gate, with pins extending from the body of the gate.

Object options:

width: The width of the object.

height: The height of the object.

angle: The rotation angle.

pinLength: The length of the pins extending from the gate.

fg: The foreground colour. Use the `Color` object to set this option.

bg: The background colour. Use the `Color` object to set this option.

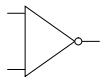


Figure E.7: NotGate object

E.2.8. Resistor

Draws a standard resistor symbol, with pins extending from the body of the symbol.
Object options:

width: The width of the object.

length: The length of the object.

angle: The rotation angle.

pinLength: The length of the pins extending from the gate.

fg: The foreground colour. Use the `Color` object to set this option.

bg: The background colour. Use the `Color` object to set this option.

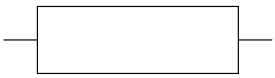


Figure E.8: Resistor object

E.2.9. Capacitor

Draws a standard capacitor symbol, with pins extending from the body of the symbol.
Object options:

width: The width of the object.

sep: The separation of the two plates of the capacitor.

angle: The rotation angle.

pinLength: The length of the pins extending from the gate.

fg: The foreground colour. Use the `Color` object to set this option.

bg: The background colour. Use the `Color` object to set this option.

Figure E.9: Capacitor object

E: Bibliography

[1] Matthew Chapman. http://www.cse.unsw.edu.au/~matthewc/pspresent/.

[2] Adobe Systems Incorporated. http://www.adobe.com.