Pyscript

Postscript Graphics with Python

v 0.4

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

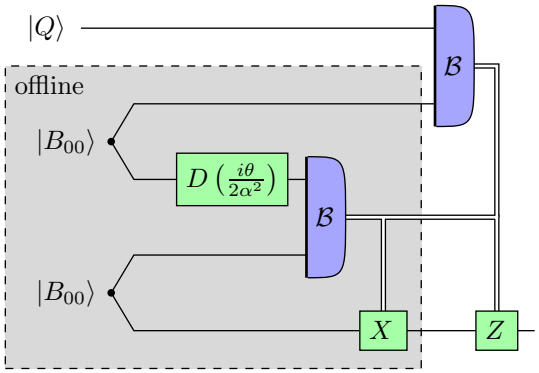
Just to be clear from the outset, some conventions follow, and some common pitfalls to be aware of ...

- 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

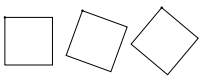
As the following example shows, the named point always stays the same in the objects coordinate system (watch the dot).

```
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

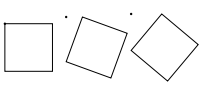
A variation of the previous example, where we'll put a dot at the nw corner of the bounding box

```
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 20 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.

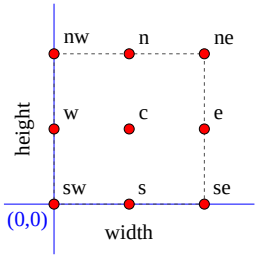
3.1.3. Area()

```
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.

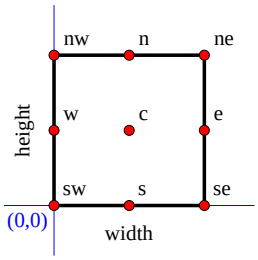
3.2.Drawing Objects

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
```

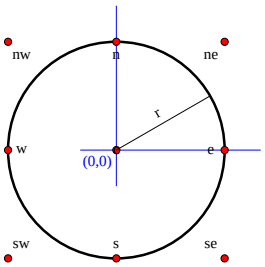
```
start=0
end=360
linewidth=defaults.linewidth
dash=defaults.dash

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
```

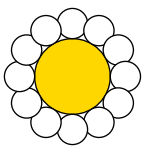
Draw a circle. The circle is specified by its position and its radius. You can also specify part of a circle with the attributes `start` and `end` which are in degrees clockwise from the top. As with the `Rectangle` there are named points on the enclosing square that corresponds to the compass points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

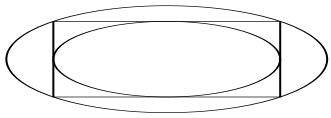
Example

```
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()

An abritrary path (line curve etc).

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=...)
```

SWEPT AWAY
SWEPT AWAY

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"$\alpha$"`.

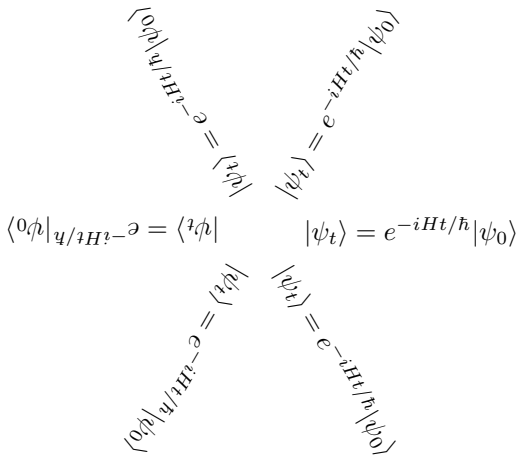
The object inherits from the `Area` object, and can also be scaled, rotated, etc. as will any of the other objects.

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.

black	burlywood	darkseagreen	lightsteelblue
dimgray	antiquewhite	lime	cornflowerblue
gray	tan	lightgreen	royalblue
darkgray	navajowhite	palegreen	navy
silver	blanchedalmond	honeydew	darkblue
lightgrey	papayawhip	seagreen	mediumblue
gainsboro	moccasin	mediumseagreen	blue
whitesmoke	orange	springgreen	midnightblue
white	wheat	mintcream	lavender
maroon	oldlace	mediumspringgreen	ghostwhite
darkred	floralwhite	mediumaquamarine	slateblue
red	darkgoldenrod	aquamarine	darkslateblue
firebrick	goldenrod	turquoise	mediumslateblue
brown	cornsilk	lightseagreen	mediumpurple
indianred	gold	mediumturquoise	blueviolet
lightcoral	lemonchiffon	darkslategray	indigo
rosybrown	khaki	teal	darkorchid
snow	palegoldenrod	darkcyan	darkviolet
mistyrose	darkkhaki	cyan	mediumorchid
salmon	olive	aqua	purple
tomato	yellow	paleturquoise	darkmagenta
darksalmon	beige	lightcyan	fuchsia
coral	lightgoldenrodyellow	azure	magenta
orangered	lightyellow	darkturquoise	violet
lightsalmon	ivory	cadetblue	plum
sienna	olivedrab	powderblue	thistle
seashell	yellowgreen	lightblue	orchid
saddlebrown	darkolivegreen	deepskyblue	mediumvioletred
chocolate	greenyellow	skyblue	deeppink
sandybrown	chartreuse	lightskyblue	hotpink
peachpuff	lawngreen	steelblue	lavenderblush
peru	darkgreen	aliceblue	palevioletred
linen	green	dodgerblue	crimson
bisque	forestgreen	slategray	pink
darkorange	limegreen	lightslategray	lightpink

Figure 3.1: Named colors

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):
```

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.

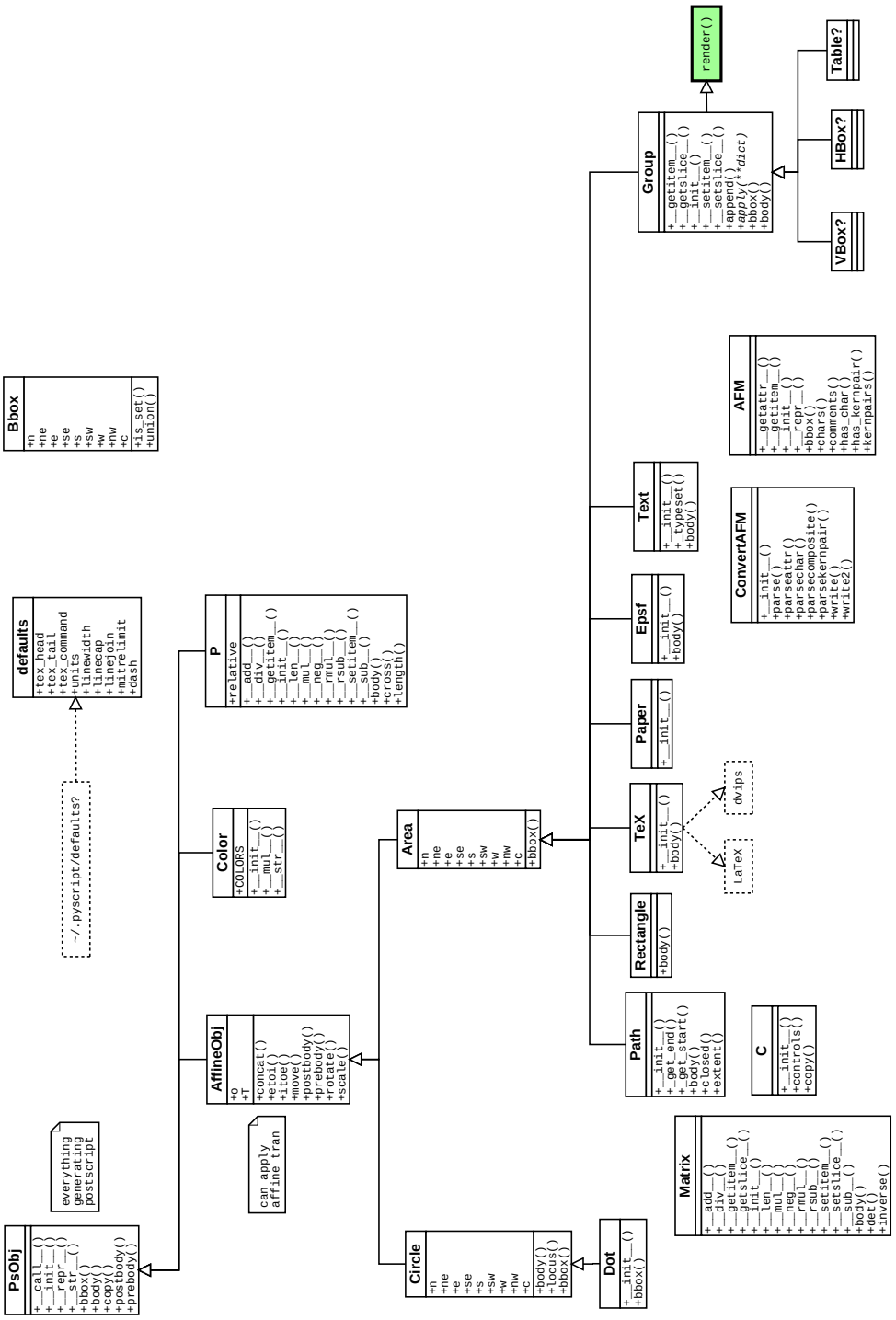


Figure 4.1: Class structure of pyscript

A: PyScript Plotting Package

B: PyScript Electronics Object Package

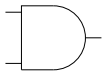
B.1.**Introduction**

Thanks to Adrian Jonstone's lcircuit macros from CTAN for the ideas and names.

B.2.**Objects**

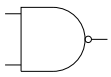
B.2.1. AND gate

```
def AndGate(
    sw=P(0,0),
    direction='e',
    height=2.0,
    width=3.0,
    pinLength=0.5,
    label=None,
    labelPinIn1=None,
    labelPinIn2=None,
    labelPinOut=None
    ):
```



B.2.2. NAND gate

```
def NandGate(
        sw=P(0,0),
        direction='e',
        height=2.0,
        width=3.0,
        pinLength=0.5,
        label=None,
        labelPinIn1=None,
        labelPinIn2=None,
        labelPinOut=None,
        ):
```



B.2.3. OR gate

```
def OrGate(
        sw=P(0,0),
        direction='e',
        height=2.0,
        width=3.0,
        pinLength=0.5,
        label=None,
        labelPinIn1=None,
        labelPinIn2=None,
        labelPinOut=None):
```

B.2.4. NOR gate

```
def NorGate(
        sw=P(0,0),
        direction='e',
        height=2.0,
        width=3.0,
        pinLength=0.5,
        label=None,
        labelPinIn1=None,
        labelPinIn2=None,
        labelPinOut=None):
```

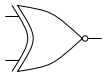
B.2.5. XOR gate

```
def XorGate(
        sw=P(0,0),
        direction='e',
        height=2.0,
        width=3.0,
        pinLength=0.5,
        label=None,
        labelPinIn1=None,
        labelPinIn2=None,
        labelPinOut=None):
```

B.2.6. NXOR gate

```
def NxorGate(
        sw=P(0,0),
```

```
        direction='e',
        height=2.0,
        width=3.0,
        pinLength=0.5,
        label=None,
        labelPinIn1=None,
        labelPinIn2=None,
        labelPinOut=None):
```



B.2.7. NOT gate

```
def NotGate(
        sw=P(0,0),
        direction='e',
        height=2.0,
        width=3.0,
        pinLength=0.5,
        label=None,
        labelPinIn1=None,
        labelPinIn2=None,
        labelPinOut=None,
        ):
```

B.2.8. Resistor

```
def Resistor(
        w=P(0,0),
        direction='ew',
        resLength=3.0,
        resWidth=1.0,
        pinLength=0.5,
        label=None,
        labelPinIn=None,
        labelPinOut=None,
        ):
```

B.2.9. Capacitor

```
def Capacitor(
        w=P(0,0),
        direction='ew',
        capHeight=1.0,
        capSep=0.25,
        pinLength=0.5,
        label=None,
        labelPinIn=None,
        labelPinOut=None,
        ):
```