Reference manual for the g.py module

someonesdad1@gmail.com

Created: Dec 2001
Version: 11 Mar 2022

Change History

11 Mar 2022	Moved to the /plib repository. Tested with python 3.7.12. `Setup()` added to g.py with `portrait` and `mm` as the defaults. I'm still using the LaserJet 4050 I bought in the early 1990's, so I haven't moved this code forward to support more modern Unicode stuff.
10 Dec 2015	Minor fixes.
4 Jun 2014	I've made a decision to only support/test from python 2.6.5 and forward. In particular, I've modified the code and tested it with python versions 2.6.5, 2.7.6, and 3.4.0 and it appears to work adequately under these versions (the criterion was comparing the output of the demo scripts). Version 3.2.2 fails in a deep copy in `g.py` of the graphics state and I suspect it is a bug in python because the other versions I tested with don't exhibit this problem. Under python 3, I can't run the `cd_label.py` script because there's no PIL for python 3. I've also had Unicode encoding problems with the `font_gallery.py` and `symbols.py` scripts, so these aren't working fully under python 3.4.
Apr 2013	Minor fixes. Fixed parameter name in `RegularPolygon()`.
19 May 2012	Minor edits. Added comment about how to convert for use with python 3.
2 Dec 2011	Fixed parameter name in `GradientFill()` (was angle_in_degrees; is actually Angle in code).
Dec 2001	Original release.

What is it and why would I want to use it?

It's a python library available at https://github.com/someonesdad1/plib; it's in the directory `g`. The primary file is `g/g.py`.

The `g.py` module gives the python programmer the ability to construct graphical drawings using PostScript as the output format. The module is a thin wrapper for PostScript; marrying python and PostScript gives a good general-purpose programming environment for generating graphical output. It is a page-oriented drawing tool with the following capabilities:

- ♦ Draws elementary geometrical figures and paths. You control how they're filled and how their borders are drawn. Fill with solid colors, parallel lines, gradient colors, or bitmaps. Line colors, fill colors, and text colors are maintained independently.
- ♦ Text can be placed on the page with left, center, or right justification. You can print text along a path.
- ♦ You set orientation (e.g., portrait or landscape), drawing units (e.g., inches or mm) and paper size. If the selections included don't meet your needs, it's easy to add new ones.
- ♦ You can place bitmaps on the page using the Python Imaging Library. Supported bitmap file types include JPEG, PNG, and TIFF. You can modify the bitmaps with the PIL before placing them on the page.
- ♦ Clip to an arbitrary path. Paths can contain straight line segments, circular arcs, and Bezier

curves and can be made up of multiple disjoint regions.

- ♦ Save and restore the graphics state, either on the PostScript-like graphics state stack or in your own data structure.
- ♦ You can use coordinate transformations of scaling, rotation, and translation. These can be combined to effect arbitrary affine transformations.
- ♦ You can change the module's defaults to suit your tastes.
- ♦ Simple debugging support is provided. You also have the python debugger to help with debugging your programs.
- ♦ If you know PostScript, you can issue arbitrary PostScript commands from within your drawing code using the inline() function.

I wrote this tool in 2001 because it was the tool I wanted for my own use. The implementation is a few thousand lines of python code.

While I have a PostScript-capable printer, I don't install a PostScript driver on my computer. This is because I always convert the PostScript output to a PDF file using GSView. Using a PDF viewer, I can then print with whatever is handy (I just use the PCL5 driver on my printer, as that gives me what I need).

Limitations

The `g.py` tool has a number of limitations. To me, the biggest one is that there are no font metrics available because the library is not talking to a PostScript interpreter. This means you can't do things like cuddle text up next to an object or nicely wrap text in a box. While this probably wouldn't be terribly difficult to implement, at the time I wrote the tool I didn't want to spend the effort and create a dependency, so I left it out. I haven't missed it terribly, but it would be nice to have.

Because of the previous limitation, the current point is undefined after putting text on the page. However, you can immediately place some more text and it will append after the previously-placed text.

There are also some limitations with paths because the library isn't talking to an interpreter. A path is constructed and saved within python's data structures. The whole path will be drawn with the current join and cap types, unlike PostScript, where you can change the join and cap types while creating the path. You also cannot issue a transformation while a path exists.

There is no support for transparency. This was a fundamental limitation of PostScript when I wrote the library and still appears to be a limitation of PostScript.

PostScript also has no ability to perform raster operations with a bitmap and the drawing material already on the page (these are features that are used when doing fancy bitmap stuff).

Architecturally, I think the program is weak when compared to the design of `piddle/sping`, some defunct graphics libraries available when I wrote `g.py`. I liked the design `piddle` used of having interchangeable back-ends for different types of graphics output. I didn't have enough experience with graphics stuff at the time to do this.

Strengths

The library has been essentially unchanged since 2001. If I recall correctly, I developed it with python 2.6.5. It should work indefinitely into the future as long as python's basic semantics and syntax don't change.

I have used the library to create many hundreds of drawings over the years and it has served my needs well. I've found that virtually all of my use of this library involves using the simple graphical elements: lines, rectangles, circles and placing text on the page. I also use translations and rotations constantly, dilatations (scaling) a bit less, and very rarely any of the fancier features I added.

Bugs

5 Oct 2011: if I recall correctly, there have only been a few bugs. Somewhere around python 2.2 , something changed (not in the g library though) and the `cd_label.py` demo needed a hack. Also, a number of years ago something changed in GSview and the generated PostScript that used to work quit working. I found that if I left the calls to `gclose()` out of my code, things would work again. **However, this disables the warning that counts push/pop calls, so be forewarned**.

Frankly, I also occasionally come across corner cases that are either bugs in ghostscript or bugs in the g library; I'm not sure which (although I think it's more likely they're in the g library). I usually quickly find a workaround and I haven't documented them.

I consider stability of the code quite important, so there are a few cases where you'll get an exception or error message. `SetPageSize()` is one such case where you're told it's not implemented yet; I haven't missed the functionality.

The original `g.py` library used mixed camel-case names. Thus, for example, `SetOrientation` was `setOrientation`. In the decade since I wrote this library, I've come to dislike mixed camel case, so I modified the library to use normal camel case. You can use either form as you wish.

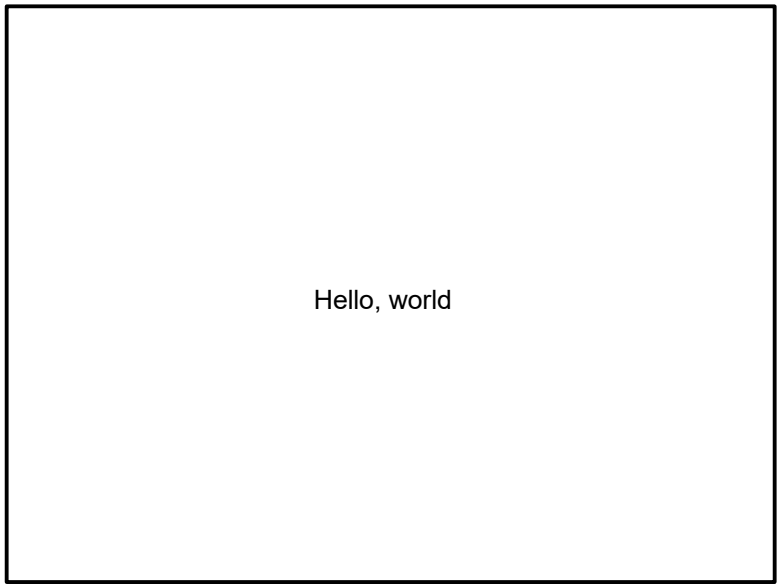
Example code

First, let's give a "hello world" type program. This code places "Hello world" on the page and draws a box around it:

```
from g import *

output_stream = open("helloworld.ps", "w")
ginitialize(output_stream)
setOrientation(portrait, inches)
move(1, 1)
rectangle(5, 4)
move(3, 3)
text("Hello, world")
```

To mentally understand how this works, `g.py` (and PostScript) use a right-handed Cartesian coordinate system with the origin at the lower left corner of the page. The page orientation is portrait, which means the short edge is along the x axis; the default units are in inches. The current drawing point is moved to (1, 1) and a rectangle of width 5 inches and height 4 inches is drawn from this point (this point is the lower left hand corner of the rectangle). The current point is moved to (3, 3) and the text is placed on the screen there. The output will look roughly as follows:



Here's a longer program that illustrates a typical task that I use the `g.py` library for. It prints a degree scale similar to what you'd see on a protractor. I removed most of the comments to make it a little more terse, but you should still be able to see what's going on.

```
from g import *
wrap_in_PJL = 0

def SetUp(file, orientation=portrait, units=inches):
    ofp = open(file, "w")
    ginitialize(ofp, wrap_in_PJL)
    setOrientation(orientation, units)
    return ofp

def DegreeScale(file):
    s = SetUp(file, portrait, inches)
    lineWidth(.01)
    textSize(.15)

    translate(4.25, 5.5)
    radius  = 3.5

    # Draw circle center crosshairs and axis lines
    delta = .1
    line(-delta, 0, delta, 0)
    line(0, -delta, 0, delta)
    push()
    for i in xrange(4):
        line(2*delta, 0, radius-delta, 0)
        rotate(90)
    pop()

    # Draw tick marks every degree
    delta = 0.05  # Tick mark length
    push()        # Isolation so we can use rotate()
    for i in xrange(360):
        line(radius, 0, radius+delta, 0)
        rotate(1)
    pop()
```

```
        # Draw tick marks every 5 degrees
        push()
        for i in xrange(360/5):
            line(radius, 0, radius+1.5*delta, 0)
            rotate(5)
        pop()

        # Draw tick marks every 10 degrees and add a text label
        push()
        for i in xrange(360/10):
            line(radius, 0, radius+2*delta, 0)
            rmove(delta, -delta)
            text("%d" % (10*i))
            rotate(10)
        pop()
        gclose()
        s.close()

    DegreeScale("out/degree_scale.ps")
```

History

I wrote this library in an intensive two week period around Thanksgiving of 2001 and I was learning PostScript at the same time (a friend at work had loaned me his copy of the PostScript red book). I had been wanting a general-purpose graphical library for e.g. generating custom scales for my shop since the 1970's. I had earlier written a python graphics script that output PCL5 (because I had a LaserJet printer that could process it), but I didn't like PCL5's graphical model and wanted something more capable (and that made vector drawings).

I was thinking about using PCL6, but decided to first take a look at PostScript, since it was another popular page description language. While PCL6 was very capable, its big disadvantage was that there were no free utilities to display the output on the screen -- unlike PostScript, which had GSView on Windows and similar tools on Linux. As soon as I studied PostScript, it resonated with me because its graphical model was nearly identical to what I had envisioned in the 1980's.

I definitely wanted a vector drawing language and there weren't many choices other than PostScript. SVG would have been a good choice, but there were no tools or good documentation available at the time and it was pretty new, so PostScript was the obvious choice.

I also examined the `piddle` library at the time and liked most of what I saw -- I especially liked the fact that you could use different back-ends to render into different formats. But `piddle` didn't handle transformations (and I didn't feel like writing some wrapper code) and I didn't care for the user interface of the library. It appears `piddle` has gone defunct and the project that replaced it, `sping`, has also gone defunct.

The original design was an object-oriented design, but my implementation became hard to maintain and I felt that the object usage might be confusing to a new user (this might have also been due to my inexperience with designing and writing object-oriented programs). Thus, I changed the implementation to use a bunch of function calls. From a user's standpoint, there's no strong distinction between using functions or object methods (there's still a whole mess of them and you continually have to look them up in the reference manual). An object design would have been valuable to allow later subclassing for specialized applications, but I haven't really needed that, so the imperative approach has been adequate.

My workflow

While using `from x import *` is a poor python programming practice in general, I use it constantly in my scripts that generate something graphical. In practice, I've found no namespace collisions of note with the other modules I use on a regular basis (mostly numpy).

When I start a new script to generate some graphical output, I include the following function to set up the `g.py` environment the way I want it:

```
def SetUp(file, orientation=landscape, units=inches):
    '''Convenience function to set up the drawing environment and return a
    file object to the output stream.
    '''
    ofp = open(file, "w")
    ginitialize(ofp, wrap_in_PJL=0)
    setOrientation(orientation, units)
    return ofp
```

I call this `SetUp` function, then start writing graphical code immediately. Since I work in two bash shell windows on my screen, one has the script open in my editor and the other is used to run the script with a one-letter command. I name the output file something ending in `.ps` to indicate a PostScript file. On Windows, I launch this PostScript file with its default program, which is GSView. I have GSView set up to automatically refresh its drawing when the PostScript file changes, so all I have to do is edit the script, move to the other window and run the one-letter command, then move the mouse to the GSView window and I'll see the updated drawing. Since modern PCs are so fast, this all happens essentially instantaneously unless the drawing is complicated. On Linux, I convert it to a PDF with `ps2pdf` and view it with the default PDF client.

This gives me a very quick code-test-fix cycle. Even though the `g.py` library (and PostScript) are fairly low-level tools, I usually get where I want to go fairly quickly.

What you need to use it

The only required tool is a python installation; I would imagine that the `g.py` module should work with any late-model python 3 version. I've used it quite a bit with python 3.7, which I've stuck with for a number of years.

You can get python from http://www.python.org.

Python Imaging Library: If you want to use the `picture` command, which puts a bitmap on the page, then you must have the Python Imaging Library (PIL) installed. PIL isn't available on python 3, but if you need to use it, you may want to search out a library called `pillow` (a fork of PIL) and see if that can work for you (I haven't tried it). The PIL versions for python 2.6 and 2.7 work with the `g.py` script.

Demo files

A number of demonstration files are included in the `demo` directory:

`4x6_card.py`	Prints a graph-paper pattern on a 4x6 index card (I make these for sketching things in my shop).
`bus_card_angle_measure.py`	Makes a card you can print to help estimate distances to things. Read the associated PDF file.
`cal_plain.py`	Calendars
`calendar.py`	
`cd_label.py`	Prints text and an image on a CD label. Requires the Python Imaging Library to print the image.

center_contrast.py	Display the two images side-by-side; the center color is the same in both cases.
color_tables.py	Prints some tables for looking up a color name.
colored_boxes.py	Randomly-colored boxes.
degree_scale.py	Prints a degree scale similar to a compass rose.
ellipses.py	Some randomly located, colored, and sized ellipses.
eng_grid.py	Various engineering grids that can be printed on a printer. Scaled in inches to letter-size paper, so metric/A4 folks will have to hack on it a bit to get what they want.
fill_eofill.py	Shows the difference between the two filling algorithms used in the program.
filled_path.py	Demonstrates some filled paths.
font_gallery.py	Shows fonts available for my default printer (a LaserJet 4050).
gradient_fill.py	Demonstrates some gradient fills -- they're coarse enough to let you see the algorithm.
grid.py	An 8x10 inch grid on letter-size paper.
helloworld.py	Probably the first demo script you should run to make sure things are working properly.
hsv_boxes.py	Hue-saturation-value boxes.
illusion.py	An optical illusion.
number_spiral.py	Integers plotted on an Archimedean spiral; the primes are highlighted.
paper_scale.py	A scale I once made to measure the physical margins of my laser printer.
piddle_demo.py	Duplicates the demo included with the `piddle` package, a defunct python graphics library.
radial_text.py	Text printed in various radial directions.
resolution_target.py	Duplicates a 1951 USAF-type of resolution target.
rounded_rect.py	Shows rounded rectangles.
symbols.py	Characters available in the symbol font on my printer.
target_bb.py	Targets for BB guns.
target_rifle.py	Targets for rifles.
text_path.py	Examples of printing text along paths.
white.py	Another optical illusion.

Graphics model

The graphics model is essentially that of PostScript. The origin of the page is in the lower left corner and the coordinate system is a right-hand Cartesian system. The default length unit is the point (as it is in PostScript). I use `g.Setup()` and use inches or mm as the default units.

The ability to transform the coordinate system with translations, rotations, and dilatations give lots of power. This is combined with the `g.py`'s commands of `push()` and `pop()`, which save the graphics state on a stack and allow you to make changes which don't affect the previous pushed state.

All references to angles in this library are measured in degrees. This was somewhat of a daring

design decision when I did it because I'm so used to programming trig functions in radians, but it has stood the test of time.

Components of the graphics state

When you are using the `g.py` module, you'll want to keep a mental picture of the things that make up the current graphics state. The attributes of the graphics state are:

1. Paper size selected
2. Orientation
3. Units mapped onto the paper
4. Line width
5. Line color
6. Line type (solid or dashed)
7. Line cap type
8. Line join type
9. Line drawing on or off
10. Fill type (solid, line fill, or gradient fill)
11. Fill color
12. Gradient fill second color
13. Fill line type (solid or dashed)
14. Filling on or off
15. Line fill angle
16. Line fill separation
17. Line fill phase
18. Text typeface
19. Text size
20. Text color
21. Coordinate transformation matrix
22. Current path
23. Current clipping path
24. Current point

There may be other attributes in the graphics state dictionary, but these are for the internal use of the module.

The modules

The `g.py` module contains the library functions that the user calls. The `go.py` file contains support code for the `g.py` file.

The `gco.py` file contains constants that maps names to integers. You can read through this file to see what things are supported, like paper sizes, dimensional units, font names, color names, etc.

Run `gco.py` as a script and it will write a file `gco.constants` showing the number values of the variables like `inches` or `mm`. You can also search for matching regular expressions for the names.

Initialization and changing the graphics state

ginitialize(stream=sys.stdout, wrap_in_PJL=0)

Used to initialize the `g.py` module. If `wrap_in_PJL` is true, the output will be wrapped so as to be a well-defined PJL job for submission to an HP LaserJet printer. `stream` is the stream where the PostScript commands will be sent. The user is responsible for opening and closing this stream.

SetOrientation(orientation, units=inches)

Sets one of four orientations of paper: `landscape, portrait, seascape`, or `inverse_portrait`. If you look at a piece of letter or A4 paper in the conventional way (long edge vertical), that's called portrait mode. If you rotate that page 90 degrees counterclockwise about an axis normal to the page, you'll be looking at the page in `landscape` orientation. *This assumes that the origin of the (right-handed) Cartesian coordinate system mapped onto the paper is always in the lower left corner of the page with the X axis increasing to the right and the Y axis increasing upwards.* Another 90 degree counterclockwise rotation results in `inverse_portrait` mode. One more 90 degree counterclockwise rotation puts the page into `seascape` mode. In all four cases, the coordinate system is the conventional right-handed Cartesian coordinate system and the origin is always at the bottom left-hand corner of the page.

The `units` parameter defines the drawing units that will be mapped to the page. Typical choices are `inches`, `mm`, `points`, and `cm`.

If you wish the coordinate system to be correctly placed on the paper, you must call the `SetPageSize()` function before calling `SetOrientation()` if you are using a page size that is not the default.

Defaults: `portrait`, `inches`

SetPageSize(size)

`size` is an integer defined as a constant. It determines the paper size as defined in the `paper_size` dictionary. It contains common values such as `paper_letter, paper_legal`, etc. You can edit the `g.py` file to add custom sizes to the `paper_size` dictionary. This command has no effect on the graphics state except for how it influences the `SetOrientation()` function to locate the origin of the coordinate system.

Default: `paper_letter`

NewPage()

Causes a page ejection command to be sent to the printer. The page following will be set to the current defaults. Example: if you set the first page to landscape orientation in inches and never change the orientation or units, all subsequent pages will be set to landscape orientation in inches with the origin at the lower left corner of the page after a `NewPage` command.

SetGS(new_gs)

Return to a saved graphics state by passing in a graphics state dictionary that was previously gotten from a call to `GetGS()`. You can change the values in the dictionary if you wish, but realize there is no parameter checking on the values, since that is normally done by the accessor functions provided by the `g.py` module.

GetGS()

Returns a dictionary representing the current graphics state. You can store it away and return to the saved state at any time by calling `SetGS()` with the saved state. You can accomplish a similar thing by using the `push()` and `pop()` functions.

reset()

Emits the necessary PostScript to set the PostScript interpreter to the graphics state defined by the `g.py` module's graphics state. You might use this function, for example, after you had made a call to `inline()` to send some custom PostScript to the drawing device. Executing the `reset()` function would ensure the PostScript graphics state was consistent with the `g.py` module's graphics

state..

move(x, y)

Moves the current point to the point `(x, y)`. Any floating point numbers are allowed.

rmove(x, y)

Moves a relative horizontal distance `X` and a relative vertical distance `Y` from the current point. Any floating point numbers are allowed.

push()

Saves the current graphics state by pushing it on an internal stack. You can return to the pushed state by executing the `pop()` command. After executing the `push()` command, no commands can affect the previously-saved graphics states. This allows you to embed whole drawings inside of another drawing. For example, a page drawing routine could `push()` its state, then call a subroutine to draw a logo on the page; the logo could be scaled to half its normal size, then drawn. When the code drawing the logo was finished, the calling context would execute `pop()` and it would be in the state immediately before the call to `push()`.

The `g.py` module will issue a warning to stderr when `gclose()` is called if the number of calls to `pop()` doesn't match the number of calls to `push()`. This is intended to alert you that your code might be missing a `pop()` call.

pop()

Restores a previously saved graphics state from the internal stack (saved by the `push()` command).

Line characteristics

LineCap(cap)

Determines the type of ending on a line. The allowed values are

`cap_butt`	The line is squared off at the end of the line.
`cap_round`	The end of the line is a semicircular arc with the diameter equal to the line width.
`cap_projecting`	The line is squared off at the end of the line, but it extends a half linewidth past the line's endpoint.

LineColor(color)

Sets the color that lines and borders will be drawn with. Remains in effect until another `LineColor()` call is made or `SetColor()` is called. The `color` parameter is a tuple of three numbers between 0 and 1 (the RGB values). Numerous constants are given to allow setting the colors by name rather than by having to use the RGB values.

Default: `black`

LineJoin(join)

Determines how lines join when they meet at a common point (only applicable to paths). The allowed values are:

`join_miter`	The lines are mitered together, resulting in a sharp point.
`join_round`	The lines meet with a rounded corner.
`join_bevel`	The lines are joined as with a miter, then the miter is beveled off.

LineOff()

Turns off border drawing of figures and paths and makes lines drawn with `line()` or `rline()` to be invisible.

Default: line/border drawing is turned on.

LineOn()

Turns on the border drawing of figures and paths and allows lines drawn with `line()` or `rline()` to be visible.

Default: line/border drawing is turned on.

LineType(linetype)

Selects the type of line to draw. To change the proportions of the dashed lines (but not their width), use the `ScaleDash()` function. The selections are:

```
solid_line
dashed
dash_little_gap
dash_big_gap
little_dash
dash_dot
dash_dot_dot
```

Default: `solid_line`

LineWidth(Width)

Sets the line width in the current units of measure. If you want the physical width of the line to stay the same size on the paper after the `scale()` function is called, use the call `ScaleLineWidth(yes)`.

Default: `0.01`

ScaleDash(scale_factor=1)

The default dashed lines look reasonable on a regular page. However, you may want to scale them up or down by using this function. The `scale_factor` must be a floating point number greater than 0. It scales the dash pattern, but has no effect on the line width. Calling `ScaleDash()` with no parameter will result in there being no scaling of the dash patterns. Note this also will scale the dashed lines that are used in the line filling routines. Setting `scale_factor` to less than 1 will cause the dashes and spaces in the lines to get smaller.

ScaleDashAutomatically(yes_or_no)

Scales the dashed lines (both regular and fill lines) so that they appear on the output device the same before and after an isotropic change of scale.

Default: `yes`

ScaleLineWidth(yes_or_no)

If you execute a `scale()` call, the width of drawn lines will scale too (example: if you set the line width to 1 while the units on the page were points, then scaled the page to inches, your lines would be 1 inch wide). This may be unexpected behavior. The `ScaleLineWidth()` function called with a value of `yes` will make the width scale whenever an isotropic scaling transformation is applied (it will be left unchanged if the transformation is anisotropic). This means that the physical line width on the paper will be the same before and after the scale transformation. Since this behavior is anticipated to be the most commonly desired, it is the default.

Default: `yes`

Fill characteristics

FillColor(color)

Sets the color that fills will be drawn with. Remains in effect until another `FillColor()` call is made or `SetColor()` is called. The `color` parameter is a tuple of three numbers between 0 and 1 (the RGB values). Numerous constants are given to allow setting the colors by name rather than by having to use the RGB values.

Default: `black`

FillOff()

Turns the filling of closed figures off. The figures will appear as if they are transparent, allowing the underlying drawing elements to show through.

Default: filling is turned off.

FillOn()

Turns on the filling of closed figures, both closed paths and the built-in objects such as rectangles, circles, etc. The fill will be drawn with the current fill color and type.

Default: filling is turned off.

FillType(fill_type)

Determines the type of filling to perform on figures and closed paths. The allowed values are: `solid_fill, line_fill`, and `gradient_fill`.

Default: `solid_fill` (however, filling is turned off by default)

GradientFill(bottom_color, Angle=0.0, factor=1.0)

Sets the second color to be used with gradient fills and the angle of the fill. When a gradient fill is specified, it will go from the currently set fill color at the top to `bottom_color` at the bottom. The method of producing the fill is to draw a number of boxes whose color is linearly interpolated in RGB space between the two colors. The program calculates how many boxes to draw. If you want to adjust this number of boxes, you can make `factor` a number different than 1. For example, if you see objectionable banding in the output, you could double the number of boxes drawn by setting `factor=2`. `Angle` is the angle of the fill in degrees.

If you examine the PostScript output with a tool such as GSView, you may see numerous lines or dashed lines on the screen in the filled regions. These are artifacts of rendering to the screen; you shouldn't see these artifacts in a printed output. If you do, try making `factor` greater than 1.0.

Default: `white`

LineFill(angle_in_degrees, separation=0, phase=0)

When a line fill is used, this function controls how the lines fill figures or closed paths. `angle_in_degrees` is the angle the fill lines make with the current X axis. `separation` is the normal (90 degrees to the line) distance between the lines. `phase` is a decimal fraction between 0 and 1 that specifies what fraction of the separation should be used to shift the lines in a direction normal to the lines; this is useful when you want to fill a figure with lines of alternating type. The fill lines are drawn with the current fill line type and are drawn in the current fill color.

Default values: `angle_in_degrees` must be specified. `separation` defaults to 10 times the current line width. `phase` defaults to zero.

LineFillType(line_type)

`line_type` is the same parameter as in the `LineType()` function, but sets the line type for fill lines.

Defaults: `solid_line`

LineFillWidth(width)

Sets the width of fill lines.

Default: same as `LineWidth()`

Text characteristics

TextColor(color)

Sets the color that text will be drawn with. Remains in effect until another `TextColor()` call is made or `SetColor()` is called. The `color` parameter is a tuple of three numbers between 0 and 1 (the RGB values). Numerous constants are given in `gco.py` to allow setting the colors by name rather than by having to use the RGB values.

Default: `black`

TextName(font_name)

Defines the typeface that subsequent text will be drawn with. The four `font_name` values that are intended to be in every implementation are:

`Sans`	A sans-serif font such as Helvetica or Arial.
`Serif`	A serif font such as Times-Roman.
`Courier`	The fixed-spacing Courier font.
`Symbol`	For Greek letters and mathematical symbols.

There may be more fonts available to you, but they are implementation-dependent. The `g.py` module comes with constants and definitions for the PostScript fonts in the HP 4050 LaserJet printer (because that's the printer I use). You may want to change the constants and definitions to be able to work with the fonts in your printer.

Default: `Sans`

TextSize(font_size)

Sets the size of the fonts to be drawn in the current units. If you perform a subsequent scaling, the font size will be the same number, but in the new units. If you wish the size of the font to remain invariant under a scale transformation (i.e, remain the same physical size on the paper), see the function `ScaleTextSize()`.

Default: `0.15` inches (about 11 points)

ScaleTextSize(yes_or_no)

Sets the state so that any change in scale will result in the font size being changed too. This means that the physical height of the line on the page is the same before and after scaling commands. For isotropic transformations, it means you'll see the same size font printed on the page before and after the transformation.

Default: `yes`

Transformations

If you are in the process of defining a path with the path commands and you issue a call to one of the following transformation functions, you will get an exception. This is because the path object you are constructing is stored in a python data structure (for rendering into PostScript later), but the transformation functions result in PostScript getting generated immediately. Thus, to execute a transformation, you'll need to execute `NewPath()` to delete the current path.

These transformations, in general, don't commute, as you can see by the commutators of the Lie generators $\partial_x, x\partial_x, y\partial_y,$ and $x\partial_y - y\partial_x$.

rotate(angle_in_degrees)

Transform the coordinate system by a rotation about the current origin. If `angle_in_degrees` is positive, the angle is counterclockwise (i.e., the conventional mathematical direction). `angle_in_degrees` may be any floating point number.

scale(xfactor, yfactor=None)

Transform the coordinate system by a scale transformation. A call of `scale(2, 2)` means a line drawn from (0, 1) to (1, 1) in the scaled coordinate system will have a length of 2 in the coordinate system before the transformation. Neither `xfactor` nor `yfactor` are allowed to be zero. However, they can be negative, which results in reflections about the coordinate axes. If you use reflections, you will get mirror-image text (which you may or may not have intended). If `yfactor` is None, then the scaling transformation is isotropic.

translate(x, y)

Transform the coordinate system by a translation of the current origin. The new origin will be at the point `(x, y)` in the current coordinates. `x` and `y` can be any floating point values.

Paths

Note: if you are having problems getting your paths to work correctly, you can see what's in a path by printing it: `print p` (where `p` is a path gotten via `GetPath()`) will display the items that make up the path.

GetPath()

Return a copy of the internal data structure that holds the current path. You can later set the current path with this saved path by executing the `SetPath()` function.

NewPath()

Discards the current path. Any objects you add with `PathAdd()` will be added to the new path.

PathAdd(tuple_or_list, type=path_point)

Adds items to the current path. The details are:

`type`	tuple_or_list
`path_point`	A tuple of the form `(x, y)`. Adds the indicated point to the path.
`path_point`	A list of points of the form `[(x1, y1), ..., (xn, yn)]`. Has the same effect as multiple calls with a single point.
`path_arc_ccw`	A tuple of the form `(x, y, r, start_angle_in_degrees, end_angle_in_degrees)`. The point `(x, y)` is the center of the circle, `r` is the radius, and the arc goes from `start_angle_in_degrees` to `end_angle_in_degrees` in a counterclockwise direction. A line will be drawn from the current end point of the path to the first point of the arc if these two points do not coincide.
`path_arc_cw`	The same as `path_arc_ccw`, except the arc is drawn from the starting angle to the ending angle in a clockwise direction.
`path_bezier`	A tuple of the form `(x1, y1, x2, y2, x3, y3)`. Draws a Bezier from the current point `(x0, y0)` to the point `(x3, y3)`. The middle two points are the control points. A Bezier curve is a parameterized curve where the parametric equations are cubic polynomials in the parameter. The curve traces a path from `(x0, y0)` to `(x3, y3)` as the parameter ranges between `0` and `1`.

Note: the first item you add to each subpath cannot be a Bezier curve. The reason is because the first point of a Bezier in PostScript is the current point. If you had ended a subpath and started a new one, the current point would effectively be undefined.

Since the `g.py` module is a thin layer over PostScript, it inherits PostScript's behavior. Because of this, if you construct a path and issue a coordinate transformation, you will get an exception. This is because the sending of the path elements to PostScript is deferred, but coordinate transformations are sent immediately. You can disable this behavior if you wish by changing the `error_xfm_path` variable to be `no` in the `g.py` module (however, this isn't recommended).

PathAddPoint(x, y)

A convenience function; same as `PathAdd((x, y), path_point)`.

PathClose()

In the construction of the current subpath, you can use the `PathClose()` function to close the current path. This means a line is drawn from the last point entered to the first point entered into the subpath. If you call `PathAdd()` again, you will be adding points, arcs, or curves to a new subpath in the current path.

PathMove(x, y)

Starts a new subpath in the current path (and leaves the previous subpath open).

SetPath(p)

Restores the current path to a saved path `p`. The parameter `p` must have been previously gotten from a call to `GetPath()`.

Clipping

Clipping is the act of defining a path; the subsequent drawing commands are limited to putting marks on the paper inside that path. This is useful when you want to limit drawing to a particular area, but don't want to figure out whether something should be shown on the page or not -- clipping lets the PostScript interpreter figure that out.

clip(path=None)

Sets the clipping path. If the `path` parameter is `None`, the current path is used to set the clipping path; the current path is erased after this operation. Otherwise, if `path` is a path that was gotten via the `getPath()` function, the clipping path is set with this path. The method for determining whether a point is inside the path is the non-zero winding rule (see the Glossary for a description).

ClipRectangle(x0, y0, x1, y1)

A convenience function that has the same effect as calling `clip()` with a rectangular path.

eoclip(path=None)

Same as `clip()` except the determination of points inside the path is made according to the even-odd rule, rather than the default non-zero winding rule (see Glossary for a description of these rules).

unclip()

Removes the current clipping region gotten from a previous call to `clip()`. The clipping limits will be at the limits of the page (which is effectively no clipping).

Utility functions

comment(comment, newline=no)

Inserts a PostScript comment into the output stream. This is useful to let you mark a spot in the output stream, then later inspect the resulting PostScript file. You can search for the text of your command and see the PostScript that was generated near the comment. If `newline` is `yes`, then a newline is written to the stream before writing the comment; this can put an empty line in the output PostScript that can make it easier to get to the line with your editor.

Debug(str)

Allows you to print debugging strings to the debug stream. Causes no output if debugging is off. The debugging stream is `stderr` by default, but you can change it to anything you wish.

DebugOff()

Turns debugging off. Any further calls to `Debug()` will not produce any output to the debugging stream, nor will the `g.py` module print a debug message for each of its functions that are called.

DebugOn(debug_stream=sys.stderr)

Turns on debugging so that debugging information will be sent to the `debug_stream` stream via the `Debug()` call. The `g.py` module does not make any calls to `Debug()`, so any messages you see are from your calls to `Debug()`.

DumpGS()

Dumps the graphics state dictionary to the debugging stream.

gray(value=0.0)

Synonym for the `grey()` function.

grey(value=0.0)

Returns a color tuple that represents a grey value. `0.0` represents black and `1.0` represents white.

hsv2rgb(h, s, b)

Converts hue `h`, saturation `s`, and value (brightness) `v` to a tuple of RGB (red, green, blue) numbers to set a color. The parameters range from 0 to 1. Hue selects the color and saturation determines how much of the color is there (1 is the intense color, 0 is grey). A value of 1 is fully bright; 0 is fully dark.

inline(str)

Allows you to insert raw PostScript in the string `str` into the output stream. This could be used, for example, to define PostScript functions to perform special drawing tasks. However, if you use the `inline()` function, you could make the PostScript graphics state get unsynchronized from the `g.py` module's graphics state.

rgb2hsv(color)

Converts an RGB color tuple to a tuple `(h, s, v)` where `h` is hue, `s` is saturation, and `v` is value (brightness). The returned numbers are between 0 and 1.

setColor(color)

Sets the color that all subsequent drawing objects will be drawn with. This command is equivalent to the successive calls `LineColor(color)`, `FillColor(color)`, `TextColor(color)`.

Trace(str, indented=yes)

Prints a string to the tracing stream; the string is indented by the current indent. If `indented` is set to `no`, the string is not indented.

traceOff()

Turns off tracing.

traceOn(trace_stream=sys.stderr)

Turns on tracing. This will show you the calls made in the `g.py` module. This functionality is probably not all that useful for an end user, but it is helpful for someone who is debugging the `g.py` module's functionality. A more modern implementation would use decorators and python's introspection capabilities, but such things weren't available (or I didn't understand them) when the library was first written.

Functions that put marks on paper

For the functions in this section, lines or borders will be drawn in the current line color and line style if `LineOn()` has been called for the current graphics state. Closed figures will be filled if `FillOn()` has been called for the current graphics state.

arc(diameter, start_angle_degrees, stop_angle_degrees)

Draws a circular arc with the current point as center and diameter `diameter`. The arc starts at angle `start_angle_degrees` and goes in a counterclockwise direction to `stop_angle_degrees`.

circle(diameter)

Draws a circle with diameter `diameter` whose center is the current point.

ctext(string)

Draws text that is center-justified at the current point. The current point is undefined after the command.

DrawPath(p=None)

Draws the passed in path or the current path if `p` is `None`. The current path is erased after the call. If you wish to draw and fill the current path, call `FillPath(p)`, `DrawPath(p)`. or use

```
push()
FillPath()
pop()
DrawPath()
```

ellipse(major_diameter, minor_diameter)

Draws an ellipse with major diameter `major_diameter` and minor diameter `minor_diameter`. The current point is the center of the ellipse.

EllipticalArc(major_diameter, minor_diameter, start_angle_degrees, stop_angle_degrees)

Draws an elliptical arc with the current point as center, major diameter `major_diameter` and minor diameter `minor_diameter`. The arc starts at angle `start_angle_degrees` and goes in a counterclockwise direction to `stop_angle_degrees`.

EofillPath(p=None)

Fills the passed in path or the current path if `p` is `None`. The current path is erased after the call. The filling method is even-odd. Refer to the PostScript reference manual [1] for a discussion of the fill method. If you wish to draw and fill the current path, call `push()`, `EofillPath()`, `pop()`, `DrawPath()`.

FillPath(p=None)

Fills the passed in path or the current path if `p` is `None`. The current path is erased after the call. The filling method is non-zero winding. Refer to the PostScript reference manual [1] for a discussion of the fill method. If you wish to draw and fill the current path, call `push()`, `FillPath()`, `pop()`, `DrawPath()`.

line(x0, x0, x, y)

Draws a line from the point `(x0, y0)` to `(x, y)`. The current point will be `(x, y)` after the line is drawn.

picture(image_object, width, height, stretch=no)

Puts a picture on the page in a rectangular area starting at the current point and extending for width and height. This is accomplished through the Python Imaging Library (PIL), which you must have installed in order to use this function. If `image_object` is a string, it is the name of the file containing the image to use. Otherwise, it is expected to be an image object constructed by the PIL's `Image()` constructor. If `stretch` is set to `yes`, then the image is stretched to fit into the indicated rectangular area. Otherwise, the PIL will center the image in the rectangular area without distortion. The image file formats supported are those supported by the PIL, which include formats such as JPEG, PNG, and TIFF. You can get the PIL from http://www.pythonware.com/products/pil.

rectangle(width, height)

Draws a rectangle from the current point `(x, y)` to the point `(x+width, y+height)`. The current point will be unchanged after the rectangle is drawn.

RegularPolygon(diameter, num_sides, start_angle=0, draw=yes)

Draws a regular polygon centered at the current point. `diameter` is the diameter of the circumscribed circle. `num_sides` must be greater than 2. `start_angle` is the angle that the first point is rotated counterclockwise from the X axis and is in degrees. If `draw` is `yes`, the polygon will be drawn. The path constructed is returned.

rline(x, y)

Draws a line from the current point to the point `(x, y)`. The current point will be `(x, y)` after the line is drawn.

RoundedRectangle(width, height, corner_diam)

Same as `rectangle()` except the corners are rounded. The current point will be unchanged after the rectangle is drawn.

rtext(string)

Draws text that is right-justified at the current point. The current point is undefined after the command.

stext(string)

Draws text as the `text()` function. However, escaped hex values are printed in the Symbol font in the current font size. Thus, `stext("\\e3")` will print the © character.

text(string)

Draws text starting at the current point. ***The current point is undefined after drawing the text*** (this is because the `g.py` module has no way to query a PostScript interpreter for the width of the drawn text). However, if you immediately make another call to `text()`, the new text will be properly positioned at the end of the previously-drawn text. This allows you to, for example, change the color or font style of a word in the middle of a sentence.

TextCircle(string, diameter, center_angle_degrees=90, inside=no)

Draws the text along a circular path centered at the current point. The text will be justified around the `center_angle_degrees` value. The text is plotted on the outside of the circle unless `inside` is `yes`.

TextFraction(numerator, denominator)

Convenience function that places two numbers at the current text point as a nicely formatted fraction. The two parameters may be either strings or integers. This function will only work with left justified strings (i.e., do not use it in conjunction with `ctext()` and `rtext()`).

TextLines(lines, spacing=0)

Places lines of text starting at the current point and going down in steps of `spacing`. If `spacing` is 0, then the text height is used as the spacing. If `spacing` is < 0, the lines go up the page. `lines` is expected to be a sequence of strings.

TextPath(string, path, offset=0)

Draws text along a path. You can get the path from the `GetPath()` function. The `offset` parameter is the distance to move along the path before starting the text. You may get irregular text spacing if the path has sharp bends.

Constants

Colors

These colors were defined from the usual `rgb.txt` file I had on the Linux system I was using at the time I wrote this. Note that there's no absolute standard; if you aren't getting the exact color you want, you can always change the definitions in the library's file. See http://en.wikipedia.org/wiki/X11_color_names for more details.

aliceblue	coral	gold4	lightsteelblue	paleturquoise
antiquewhite	coral1	goldenrod	lightyellow	palevioletred
aquamarine	coral2	goldenrod1	limegreen	papayawhip
aquamarine1	coral3	goldenrod2	linen	peachpuff
aquamarine2	coral4	goldenrod3	magenta	peru
aquamarine3	cornflowerblue	goldenrod4	magenta1	pink
aquamarine4	cornsilk	green	magenta2	pink1
azure	cornsilk1	green1	magenta3	pink2
azure1	cornsilk2	green2	magenta4	pink3
azure2	cornsilk3	green3	maroon	pink4
azure3	cornsilk4	green4	maroon1	plum
azure4	cyan	greenyellow	maroon2	plum1
beige	cyan1	honeydew	maroon3	plum2
bisque	cyan2	honeydew1	maroon4	plum3
bisque1	cyan3	honeydew2	mediumaquamarine	plum4
bisque2	cyan4	honeydew3	mediumblue	powderblue
bisque3	darkgoldenrod	honeydew4	mediumforestgreen	purple
bisque4	darkgreen	hotpink	mediumgoldenrod	purple1
black	darkkhaki	indianred	mediumorchid	purple2
blanchedalmond	darkolivegreen	ivory	mediumpurple	purple3
blue	darkorange	ivory1	mediumseagreen	purple4
blue1	darkorchid	ivory2	mediumslateblue	red
blue2	darksalmon	ivory3	mediumspringgreen	red1
blue3	darkseagreen	ivory4	mediumturquoise	red2
blue4	darkslateblue	khaki	mediumvioletred	red3
blueviolet	darkslategray	khaki1	midnightblue	red4
brown	darkslategrey	khaki2	mintcream	rosybrown
brown1	darkturquoise	khaki3	mistyrose	royalblue
brown2	darkviolet	khaki4	moccasin	saddlebrown
brown3	deeppink	lavender	navajowhite	salmon
brown4	deepskyblue	lavenderblush	navy	salmon1
burlywood	dimgray	lawngreen	navyblue	salmon2
burlywood1	dimgrey	lemonchiffon	oldlace	salmon3
burlywood2	dodgerblue	lightblue	olivedrab	salmon4
burlywood3	firebrick	lightcoral	orange	sandybrown
burlywood4	firebrick1	lightcyan	orange1	seagreen
cadetblue	firebrick2	lightgoldenrod	orange2	seashell
chartreuse	firebrick3	lightgoldenrodyellow	orange3	seashell1
chartreuse1	firebrick4	lightgray	orange4	seashell2
chartreuse2	floralwhite	lightgrey	orangered	seashell3
chartreuse3	forestgreen	lightpink	orchid	seashell4
chartreuse4	gainsboro	lightsalmon	orchid1	sienna
chocolate	ghostwhite	lightseagreen	orchid2	sienna1
chocolate1	gold	lightskyblue	orchid3	sienna2
chocolate2	gold1	lightslateblue	orchid4	sienna3
chocolate3	gold2	lightslategray	palegoldenrod	sienna4
chocolate4	gold3	lightslategrey	palegreen	skyblue

slateblue
slategray
slategrey
snow
snow1
snow2
snow3
snow4
springgreen
steelblue
tan
tan1
tan2
tan3
tan4
thistle
thistle1
thistle2
thistle3
thistle4
tomato
tomato1
tomato2
tomato3
tomato4
transparent
turquoise
turquoise1
turquoise2
turquoise3
turquoise4
violet
violetred
wheat
wheat1
wheat2
wheat3
wheat4
white
whitesmoke
yellow
yellow1
yellow2
yellow3
yellow4
yellowgreen

Font names

Sans
SansBold
SansItalic
SansBoldItalic
Serif
SerifBold
SerifItalic
SerifBoldItalic

Paper sizes

paper_letter
paper_legal
paper_ledger
paper_A4

Orientations

portrait
seascape
inverse_portrait
landscape

Units

points
inches
mm
cm
ft

Fill types

no_fill
solid_fill
line_fill
gradient_fill

Dashes

solid_line
dashed
dash_little_gap
dash_big_gap
little_dash
dash_dot
dash_dot_dot
scale_factor

Glossary

isotropic	The same in all directions. Applied to a scale transformation, it means the x and y scaling factors are the same.
anisotropic	Not the same in all directions. Applied to a scale transformation, it means the x and y scaling factors are different.
PostScript	The page description language from Adobe Systems Inc. The `g.py` module outputs PostScript commands to perform the graphics rendering. PostScript is a trademark of Adobe Systems Inc.
non-zero winding rule	Used to determine whether a point is inside of a region, which determines if it is filled or not. Draw a line from the point to infinity. Initialize a sum with 0. If a border of the region crosses the line in the clockwise direction, add 1 to the sum; otherwise, subtract 1 from the sum. If the sum is 0 at infinity, then the starting point is outside the region. For more details, consult the red book [1].
even-odd rule	Using the same sum and line concept in the non-zero winding fill algorithm, count the number of times the line crosses a border of the region (ignore the direction of the border). If the sum at infinity is even, then the starting point is outside the region.
separation	When drawing line fills, this value is the perpendicular distance between the fill lines.
phase	When drawing line fills, this value is the offset from the line position.

The value must be between 0 and separation.

References

1. PostScript Language Reference Manual, 2nd ed., Adobe Systems Inc., 1990. (Called the "red book" because of the red cover)

2. PostScript Language Tutorial and Cookbook, Adobe Systems Inc., 1985. (Called the "blue book".)