More on Functions & Intro to VPython

Chapter 6 and beyond

Functions, again

- Functions capture reusable program fragments
- Usually they define an abstraction:
  - increment: `def inc(x): return x+1`
  - scale: `def scl(x, f): return x*f`
- Of course, functions usually capture much more interesting computations

Functions and Style

- Functions should be used carefully so that program meaning is made clearer
- Overuse of functions can obfuscate, or obscure, the program meaning
- Try to use a consistent naming scheme
  - `in_rect`, `readPoints`, `ShowLines`
- Use direct verb names: `drawRectangle`

Functions and Style

- Parameters should be meaningful
- Parameters should capture logical varying inputs
- Avoid unrelated/unexpected/surprising side-effects

Functions and Style

- Avoid undergeneralizing: try to make a function broadly reusable
- Avoid overgeneralizing: Swiss-Army Chainsaws are dangerous

Default arguments

- Python functions can have default arguments, evaluated in the defining scope

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```


Default arguments

- The default is evaluated just once when the function is defined

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```


Keyword arguments

- Functions can be called with arguments by keyword instead of position
- Any positional arguments must come before keyword arguments

```
def f(a, b):
    return a+b

print f(a=1,b=2)
print f(1,b=2)
```

Function arguments

- Functions can take functions as arguments
- "Higher-Order" functions

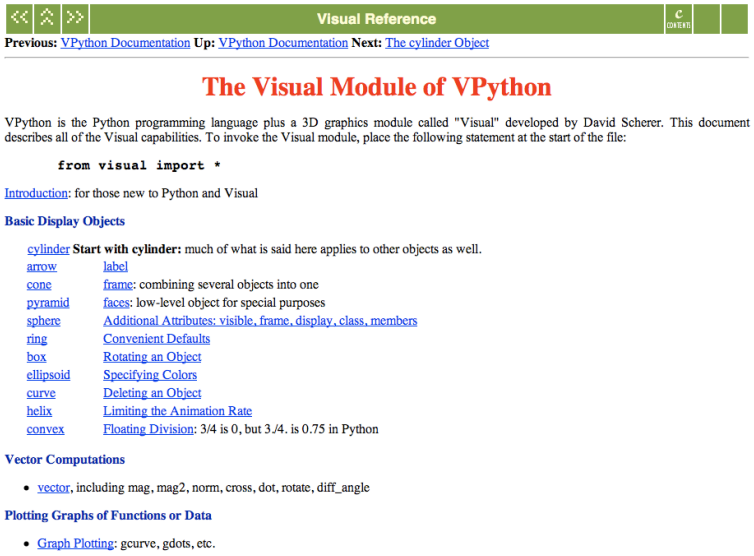
```
def f(g, x):
    return g(x)

def inc(x):
    return x+1

f(inc,2)
```


VPython

- A Python graphics module for modeling and simulation
- VPython = Python + IDLE + `visual`
- `from visual import *`



Visual Reference

Previous: VPython Documentation **Up:** VPython Documentation **Next:** The cylinder Object

The Visual Module of VPython

VPython is the Python programming language plus a 3D graphics module called "Visual" developed by David Scherer. This document describes all of the Visual capabilities. To invoke the Visual module, place the following statement at the start of the file:

```
from visual import *
```

Introduction: for those new to Python and Visual

Basic Display Objects

cylinder **Start with cylinder:** much of what is said here applies to other objects as well.
arrow — label
cone — frame: combining several objects into one
pyramid — faces: low-level object for special purposes
sphere — Additional Attributes: visible, frame, display, class, members
ring — Convenient Defaults
box — Rotating an Object
ellipsoid — Specifying Colors
curve — Deleting an Object
helix — Limiting the Animation Rate
convex — Floating Division: 3/4 is 0, but 3./4. is 0.75 in Python

Vector Computations

- vector, including mag, mag2, norm, cross, dot, rotate, diff_angle

Plotting Graphs of Functions or Data

- Graph Plotting: gcurve, gdots, etc.

VPython Visual Objects

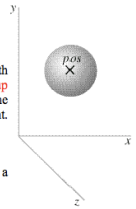
- VObjects exist for program duration
- VObjects are displayed on the display window
- VObjects have attributes: `pos`, `color`, `length`/`height`/`width`/`radius`, etc
- Changing attributes changes display

Previous: The pyramid Object **Up:** Contents **Next:** The ring Object

The sphere Object

Here is an example of how to make a sphere:

```
ball = sphere(pos=(1,2,1), radius=0.5)
```



This produces a sphere centered at location (1,2,1) with radius = 0.5, with the current foreground color.

The sphere object has the following attributes and default values, like those for cylinders except that there is no length attribute: pos (0,0,0), x (0), y(0), z(0), axis (1,0,0), color (1,1,1) which is color.white, red (1), green (1), blue (1), and up (0,1,0). As with cylinders, up has a subtle effect on the 3D appearance of a sphere. The axis attribute only affects the orientation of the sphere and has a subtle effect on appearance; the magnitude of the axis attribute is irrelevant. Additional sphere attributes:

radius Radius of the sphere, default = 1

Note that the pos attribute for cylinder, arrow, cone, and pyramid corresponds to one end of the object, whereas for a sphere it corresponds to the center of the object.

Originally there was a label attribute for the sphere object, but this has been superceded by the label object.

Visual Reference

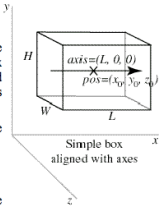
Previous: The pyramid Object **Up:** Contents **Next:** The ring Object

Previous: The ring Object **Up:** Contents **Next:** The ellipsoid Object

The box Object

In the first diagram we show a simple example of a box object:

```
mybox = box(pos=(x0,y0,z0), length=L, height=H, width=W)
```



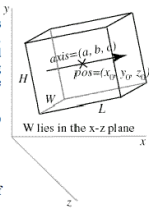
Simple box aligned with axes

The given position is in the center of the box, at (x0, y0, z0). This is different from cylinder, whose pos attribute is at one end of the cylinder. Just as with a cylinder, we can refer to the individual vector components of the box as mybox.x, mybox.y, and mybox.z. The length (along the x axis) is L , the height (along the y axis) is H , and the width is W (along the z axis). For this box, we have mybox.axis = (L, 0, 0) . Note that the axis of a box is just like the axis of a cylinder.

For a box that isn't aligned with the coordinate axes, additional issues come into play. The orientation of the length of the box is given by the axis (see second diagram):

```
mybox = box(pos=(x0,y0,z0), axis=(a,b,c), length=L,
     height=H, width=W)
```

The axis attribute gives a direction for the length of the box, and the length, height, and width of the box are given as before (if a length attribute is not given, the length is set to the magnitude of the axis vector).



W lies in the x-z plane

There remains the issue of how to orient the box rotationally around the specified axis. The rule that Visual uses is to orient the width to lie in a plane perpendicular to the display "up" direction, which by default is the y axis. Therefore in the diagram you see that the width lies parallel to the x-z plane. The height of the box is oriented perpendicular to the width, and to the specified axis of the box. It helps to think of length initially as going along the x axis, height along the y axis, and width along the z axis, and when the axis is tipped the width stays in the x-z plane.

You can rotate the box around its own axis by changing which way is "up" for the box, by specifying an up attribute for the box that is different from the up vector of the coordinate system:

```
mybox = box(pos=(x0,y0,z0), axis=(a,b,c), length=L,
     height=H, width=W, up=(q,r,s))
```

With this statement, the width of the box will lie in a plane perpendicular to the (q,r,s) vector, and the height of the box will be perpendicular to the width and to the (a,b,c) vector.

The box object has the following attributes and default values, like those for cylinders: pos (0,0,0), x (0), y(0), z(0), axis (1,0,0), length (1), color (1,1,1) which is color.white, red (1), green (1), blue (1), and up (0,1,0). Additional box attributes:

Other VPython Objects

- These are not displayed
- A vector object supports the usual: mag, mag2 (mag squared), norm (normalized), cross, dot, rotate, etc

Previous: Floating Division **Up:** Contents **Next:** Graph Plotting

The vector Object

The vector object is not a displayable object but is a powerful aid to 3D computations. Its properties are similar to vectors used in science and engineering. It can be used together with Numeric arrays. (Numeric is a module added to Python to provide high-speed computational capability through optimized array processing. The Numeric module is imported automatically by Visual.)

```
vector(x,y,z)
```

Returns a vector object with the given components, which are made to be floating-point (that is, 3 is converted to 3.0).

Vectors can be added or subtracted from each other, or multiplied by an ordinary number. For example,

```
v1 = vector(1,2,3)
v2 = vector(10,20,30)
print v1+v2 # displays (11 22 33)
print 2*v1 # displays (2 4 6)
```

You can refer to individual components of a vector:

v2.x is 10, v2.y is 20, v2.z is 30

It is okay to make a vector from a vector: vector(v2) is still vector(10,20,30).

The form vector(10,12) is shorthand for vector(10,12,0).

A vector is a Python sequence, so v2.x is the same as v2[0], v2.y is the same as v2[1], and v2.z is the same as v2[2].

```
mag( vector ) # calculates length of vector
mag(vector(1,1,1)) # is equal to sqrt(3)
mag2(vector(1,1,1)) # is equal to 3, the magnitude squared
```

You can also obtain the magnitude in the form v2.mag. and the square of the magnitude as v2.mag2.

It is possible to reset the magnitude or the magnitude squared of a vector:

```
v2.mag = 5 # sets magnitude of v2 to 5
v2.mag2 = 2.7 # sets squared magnitude of v2 to 2.7
```

You can reset the magnitude to 1 with norm():

http://www.vpython.org/VPython_Intro.pdf

```
from visual import *
ball = sphere(pos=(-5,0,0), radius=0.5, color=color.red)
wallR = box(pos=(6,0,0), size=(0.2,12,12), color=color.green)
wallL = box(pos=(-6,0,0), size=(0.2,12,12), color=color.green)
dt = 0.05
ball.velocity = vector(2,1.5,1)
bv = arrow(pos=ball.pos, axis=ball.velocity,
color=color.yellow)
ball.trail = curve(color=ball.color)
while 1:
    rate(100)
    ball.pos = ball.pos + ball.velocity*dt
    if ball.x > wallR.x:
        ball.velocity.x = -ball.velocity.x
    if ball.x < wallL.x:
        ball.velocity.x = -ball.velocity.x
    bv.pos = ball.pos
    bv.axis = ball.velocity
    ball.trail.append(pos=ball.pos)
```


VPython Graphs

- VPython has a powerful graph plotting subsystem
- `from visual.graph import *`
- Lines (gcurve), marks (gdots), bars (gvbars, ghbars), bins (ghistogram)

Graph Plotting

In this section we describe features for plotting graphs with tick marks and labels. Here is a simple example of how to plot a graph:

```
from visual.graph import * # import graphing features


funct1 = gcurve(color=color.cyan) # a connected curve object


for x in arange(0., 8.1, 0.1): # x goes from 0 to 8
    funct1.plot(pos=(x,5.*cos(2.*x)*exp(-0.2*x))) # plot
```

Importing from visual.graph makes available all Visual objects plus the graph plotting module. The graph is autoscaled to display all the data in the window.

A connected curve (gcurve) is just one of several kinds of graph plotting objects. Other options are disconnected dots (gdots), vertical bars (gvbars), horizontal bars (ghbars), and binned data displayed as vertical bars (ghistogram; see later discussion). When creating one of these objects, you can specify a color attribute. For gvbars and ghbars you can also specify a delta attribute, which specifies the width of the bar (the default is delta=1.).

You can plot more than one thing on the same graph:

```
funct1 = gcurve(color=color.cyan)
funct2 = gvbars(delta=0.05, color=color.blue)

for x in arange(0., 8.1, 0.1):

    funct1.plot(pos=(x,5.*cos(2.*x)*exp(-0.2*x))) # curve
    funct2.plot(pos=(x,4.*cos(0.5*x)*exp(-0.1*x))) # vbars
```

In a plot operation you can specify a different color to override the original setting:

```
mydots.plot(pos=(x1,y1), color=color.green)
```

When you create a gcurve, gdots, gvbars, or ghbars object, you can provide a list of points to be plotted, just as is the case with the ordinary curve object:

```
points = [(1,2), (3,4), (-5,2), (-5,-3)]
data = gdots(pos=points, color=color.blue)
```


Graph example

```
#! /usr/bin/env python
from visual import box, display
def visualize(grid):
    global scene
    #hide all the objects in the scene
    for object in scene.objects:
        object.visible = False
    rows = len(grid)                        # get number of rows
    cols = len(grid[0])                     # get number of cols
    for row in range(rows):
        for col in range(cols):
            # Choose the color for the box.  Colors are specified in
            # (r,g,b) notation.
            if grid[row][col] == 0:
                color = (1,0,0)             # make the box red
            else:
                color = (0,0,1)             # make the box blue
            # draw a box with position (x,y,z), where
            #    x=col (left (-1) to right (+1) displacement)
            #    y=-row (up  (+1) to down (-1) displacement;
            # and the given color.
            box(pos=(col, -row, 0), color=color)
def random_grid(n):
    from random import randint
    grid = []                               # intialize grid
    for row in range(n):
        rowlist = []                        # make an empty row
        for col in range(n):
            num = randint(0,1)              # fill out the row with
            rowlist.append(num)             # random numbers

        grid.append(rowlist)                # add the row to the grid
    return grid
```

```
if __name__ == "__main__":
    from visual import rate, color
    from visual.graph import gdisplay, gcurve
    scene = display(                        # 3D window parameters
        title="Random grid visualization",  # Window title
        autocenter=True                     # Automatically center object
    )
    gdisplay(                               # 2D graph window parameters
        title="Number of squares per color",# Window title
        xtitle="I",                         # Label on x-axis
        ytitle="Count",                     # Label on y-axis
        background=color.white,
        foreground=color.black
    )
    blue_line = gcurve(color=(0,0,1))       # Set up a blue curve
    red_line = gcurve(color=(1,0,0))        # Set up a red curve
    for i in range(10):
        grid = random_grid(20)              # Make a random 20x20 grid
        visualize(grid)                     # Visualize the grid
        count_blue = 0                      # Count the blue (=1) squares
        for row in grid:
            count_blue = count_blue + sum(row)
        count_red = 400 - count_blue        # Compute the red (=0) squares
        blue_line.plot(pos=(i, count_blue)) # Extend blue curve
        red_line.plot(pos=(i, count_red))   # Extend red curve
        rate(.5)                            # Allow at most 1/2 loop iterations
                                            #  per second
                                            #  (one frame per 2 seconds)
```