P4N - Python for Neuroscience

Release 2013.1

Jonathan Peirce

March 25, 2013

CONTENTS

Welcome to the materials for the Python for Neuroscience (and Psychology) workshop.

Contents:

CHAPTER

ONE

SCHEDULE

Tues 26 March 2013: Python programming

- **9-10am: Welcome and coffee**
- 10-11am: *Getting started*
- 11-12:30pm: *Variables and common types*
- **12:30-1:30pm: Lunch**
- 2-3pm: *Loops*
- 3-4pm: *Functions*
- 4-5pm: **exercises**

Weds 27th March 2013: PsychoPy

- 9-10am: Concepts, windows, stimuli, responses
- 10-12:30am: Build an experiment: Inhibition of Return
- **12:30-1:30pm: Lunch**
- 1:30-3pm: Build your own experiment
- 3-3:30pm: Performance issues
- 3:30-5pm: Build your own experiment

Thurs 28th March 2013: Data processing and advanced topics

- 9-10am: importing and exploring data files
- 10-12:30: arrays and plotting
- **12:30-1:30pm: Lunch**
- 2-3pm: Reusing existing code (sub-classing)
- 3-4pm: Advanced environments and contributing
- **4pm: Close**

CHAPTER

TWO

DAY 1 - PYTHON PROGRAMMING

2.1 Getting started

There are many different interfaces/editors for using Python. For simplicity, we're going to stick with the interface that PsychoPy provides for you, but if you have favourite editor feel free to use that instead.

2.1.1 Views

Open PsychoPy and go to the Coder view (you can close Builder for now). The Coder has a script editing panel at the top, an output panel beneath where outputs from your script will appear, and a shell panel next to the output panel

2.1.2 Shell

This is a great place just to try out a quick command and see what happens. You can check a little Python syntax and see the results of commands.

Let's type some commands into the shell panel and see what happens:

```
>>> a = 3
>>> b = a+2
>>> b
5
>>> b == 5
True
```

Note that in the shell if the command *returns* a value and you didn't provide anything to receive/store that value then it gets printed to the screen instead (this is not the case for scripts run from the editor).

Python functions are organised into *modules* and *packages* which we need to import to use:

```
>>> import os
```

You can find out what's in a module using the function `dir()`:

```
>>> dir(os)
```

In this shell window you can also find out what it contains by starting to type a command. Type these lines gradually, taking note of what happens when you type the '.' and the '(':

```
>>> os.getcwd()
'/Applications/PsychoPy2.app/Contents/Resources'
>>> os.chdir('..')
```

To repeat a previous command hit `Alt-P` on your keyboard. Repeat your `os.getcwd()` function to see where your current working directory is now.

Some of the items you see as a part of `os` are functions, whereas others are attributes. For instance, `getcwd` is a function. Consider the different outputs you get in the following two lines:

```
>>> os.getcwd
>>> os.getcwd()
```

The first returns the function itself (and prints it), whereas the second *runs* the function because of the parentheses, and prints what the function returns.

2.1.3 Editing Scripts

Although the shell is a handy place to check a quick command, it's often desirable to be able to repeat a set of commands from scratch. Type this into the editor window and save the script somewhere (e.g. firstScript.py):

```
a = "hello"
print a
b = ' world'
a+b
```

Note: Strings in Python can be defined using either `'` or `"`.

Switch the bottom panel of the Coder view to show the *Output* from the script. Hit the *Run* button (or press *Ctrl-R*). You might have expected to see `hello world` but it didn't appear. That's because in running scripts nothing is printed to the output unless you explicitly request it. Change the last line to `print a+b`.

2.1.4 Indentation

One of the unusual things in Python is that indentation (whitespace) is actually important. Try to use a genuine programmer's text editor and set it to insert spaces in place of tabs (it's hard to spot errors when you have a mixture of tabs and spaces). Many editors, will try to help you get indentation right.

Type the following into the editor after your other text:

```
for thisLetter in a:
    print thisLetter
    print thisLetter.upper()
    print 'done'
```

Note: `upper()` is a `method` that all strings have. Let's find out what else they have by using that `dir()` function from earlier. Add `print dir(a)` to the last line

---

Now, that probably didn't do what you expected. In Python the code that is included as part of the for-loop is indicated by the level of indentation, so `print('done')` was repeated for each repeat. If you added `print dir(a)` at the same level that would also have been repeated. Select the last few lines of code and press *Ctrl-[* to get this:

```
for thisLetter in a:
    print thisLetter
print thisLetter.upper()
print 'done'
print dir(a)
```

Now the code will print each of the letters in their lower case. Then the loop ends. `thisLetter` still exists but it isn't changing any more. It gets printed just once in upper case, followed by the other commands.

`print` can print multiple objects at once (if you insert commas), and you can suppress the line endings with a final comma. Try this:

```
for thisLetter in a:
    print thisLetter, thisLetter.upper(),
print 'end of the loop'
print 'done'
```

2.1.5 Comments

In Python comments are indicated by the # symbol.

Note: Annoyingly a mac keyboard doesn't show you where the # is, but you can get it using `Alt-3` if you're running under OS X. If you're using a British Mac keyboard under Windows you need `Ctrl+Alt+3`. Sigh!

You can start/end a multi-line comment with three double-quotes:

```
"""This is a potentially long piece of text that
will be ignored. If it occurs at the start of a
function it becomes the help for that function
"""

#I'm going to assign a variable
a = 5 #it's a pretty boring variable!
```

In the PsychoPy Coder window you can comment out lines with `Ctrl-'` and undo that with `Shift-Ctrl-'` If you forget it's listed in the `Edit` menu

2.1.6 Exercises

1. Create a new variable called `myPhrase` and give it the value `"A whole bunch of words"`. Using the dir() command, work out a way to split that into a set of separate words.
2. In the Shell panel import the module called `sys` and find a command in it that will tell you the version of Python that you're running. Find another to tell you about the platform you're running on.

2.2 Variables and common types

2.2.1 Assigning variables

You've already seen variables being assigned using =. You can also assign multiple things at once. Type these in and then check to see what each variable looks like afterwards:

```
a = b = 2
a, b = 2,3 #also, see what happens if the number of vals doesn't match
myString = "Hello World"
firstWord, secondWord = myString.split() #also try splitting by 'o'
tupleOfTwoWords = myString.split()
```

2.2.2 Types of variable

In Python almost everything is an *object*. There are a number of built-in objects that are very widely used:

- integers and floats
- booleans
- strings (and unicodes)
- tuples and lists
- dicts

You can find out what an object is using the `type()` function:

```
print type('a')
print type([1,2,3])
print type(5)
print type(True)
```

Finding out the type can be essential if things look the same when printed but aren't the same:

```
a = 5
b = '5'
print a, b
print a==b
print type(a), type(b)
```

But there are a huge number of additional objects and you can make your own too.

2.2.3 Integers and floats

Integers don't store anything after the decimal place. Beware that in Python up until version 3 (you're probably using version 2.6 or 2.7. You could find out by importing sys and finding that version attribute again) when you divided a pair of integers it gave you back an integer:

```
print 1/3 #surprise!
print 1.0/3
print 1/3.0
```

From version 3 upwards it will give a float if they don't divide equally. You can get the new style by running the rather strange command:

```
from __future__ import division
```

This is a good thing to do at the top of most scripts to avoid confusion, or get used to typing values as floats.

2.2.4 Strings

Python has fantastic string handling options. Try these methods that are attached to strings:

```
a = "hello world"
a.title()
a.split()
a.endswith('world')
len(a)
```

You can also combine strings in nice, simple ways:

```
x = 'abc'
y = 'xyz'
z = x*2+y
z
x+x.upper()
```

2.2.5 Slicing

Often you need to fetch a subset of an object, like a string or a list.

> **Warning:** If you're used to Matlab then be warned: in Python the first element of an array or a list is zero, not one. This will catch you out sooner or later!!
> a = 'nottingham' a[0] a[2:4] a[2:] a[:] a[-1] a[2:-2].upper()

You can convert between these different types of objects where they make sense:

```
int(1.5)
int('1')
str(1.5)
float(1)
```

but not where they don't:

```
float('f1')
int([1,2])
```

2.2.6 Formatted strings

Sometimes you need to combine numbers and strings. Imagine I wanted to make a filename to save my data. Maybe in my script I had a variable to store my subject name and another to store a stimulus attribute which was 10, 50, 100, 200 on different runs. I might try and save the data filename like this:

```
subj = 'jwp'
cond = 50
filename = subj+cond+".txt"
```

You get an error because cond is a number and your trying to add it to a pari of strings (subj and ".txt") and Python doesn't know what way you want them combined. You could convert cond into a string and have no error:

```
filename = subj+str(cond)+".txt"
```

but that doesn't provide much control of the formatting of the number. If you wanted a certain number of decimal places it couldn't set that. In the following the *%i*, *%f* and *%s* indicates that Python go and find a variable in the following list and insert it with the specified representation. Any other text just looks like itself(!). If you've ever used formatted string operations in C or Matlab these will make sense pretty quickly, but otherwise they could take some time:

```
"%i" %(23)
"an int:%i" %(5)
"a float:%f" %(5)
"before%i_after" %(200)
"%s.txt" %(subj) #assuming subj was still defined
filename = "%s%i.txt" %(subj, cond)
```

Now the real advantage of this format is that you can control the number of decimal places, and padding with zeros. Try these out:

```
"%04i" %(9)
"%4i" %(9)
"%.2f" %(9)
"%s%03i.txt" %(subj, cond)
"That took %f seconds" %(32.5432143)
"That took %.2f seconds" %(32.5243553)
```

These formatted strings may seem cumbersome to start with but they're very powerful when you get the hang of them (and they're roughly the same in most languages). There are many more variants on these operations but those are the main ones that you'll need.

For more see:

> http://docs.python.org/2/library/stdtypes.html#string-formatting-operations

2.3 Containers

Very often you need variables that store more than one value and keep them organised in some way. The two most common are lists and dictionaries.

2.3.1 Lists

For storing things that have a defined order:

```
a = [10,20,30]
b = ['a',1,1.0]
b.append('blah')
a.append(3.0)
```

Slicing works just the same as with strings:

```
a[0] #remember, python starts at zero
a[4] #so this won't work
a[-1] #this will
b[-1]
b[-1][-1]
```

Mathematical operators:

```
a+a #this might be a surprise
a+b
b*3
```

Other methods:

```
print dir([]) #go and explore some of the other methods of lists
print a.append(b)
print a.extend(b)
print a.index(30)
```

Python also defines a type of variable called a *tuple*, which has slightly different methods, but similar and I find it less useful:

```
x = (1,2) #note the different parentheses
print type(x)
print dir(x)
```

For those who have come from Matlab backgrounds, these lists might look like Matlab matrices, but they aren't. These aren't designed for mathematical operations. There is a similar object which *is* very much like Matlab matrices, which we'll explore when we look at *Day 3 - Analysing data in Python*.

2.3.2 Dictionaries (dicts)

At times you want to keep things with something that identifies what each element is. That's where you'll use a dict. These can be created in various ways:

```
stim1 = {'word':'red','ori':90,'duration':0.5}
#or just create it and add the entries afterwards:
stim2 = {}
stim2['word'] = 'blue'
stim2['ori'] = 90
stim2[duration'] = 0.3
print stim1['word']
```

Then you can access the contents in a similar way:

```
print stim1['ori']
print stim2['fail'] #error?
```

These are referred to as key-value pairs. Explore what some of the different dict methods do:

```
dir(stim1)
stim1.keys()
stim1.has_key('blue')
stim1.has_key('word')
```

2.3.3 Nesting objects within each other

Often containers are nested within each other. You might well have a list of dicts, or a dict containing lists etc.:

```
#a list of dicts
stimuli = [stim1, stim2, stim3]
stimuli[0]['word'] #this is stim1 because we start at zero!!
thisStimulus = stimuli[2]
thisStimulus == stim3

#a list of lists
coordinates=[[0,0], [2,3], [8,0]]
responses = [ [1, 1, 0, 0], #if your line ends in a comma you can go to the next line
    [1,1,1,0],
    [0,1,1,1]]
print responses
print responses[2][3] #the 4th entry of 3rd list (STARTS AT ZERO)

#or we could have done this
responses = []
cond1 = [1,1,0,0]
responses.append(cond1) #etc.
```

You can nest objects as deeply as you like. The limit is your own brain being able to keep track of what you're doing!

2.3.4 Referents not copies

In many programming languages when you assign one variable to another you get a *copy* of the original. That isn't true in Python; in Python, both variables *refer* to the same item. That means that if you *change* the item in-place then it will be changed for both variables:

```
print stim1
stim3=stim1
stim3['word']='banana'
print stim1
```

The same thing is an issue with lists:

```
a = [1, 0, 1, 1]
b=a
b.append(25)
print a
```

Or combinations of lists and dicts

```
stimuli[0]['colour']='giraffe'
print stim1
```

This concept that you have multiple variables refering to the same actual objects is very useful for conserving memory and to shorten some code. e.g. if you had a dictionary of 'subjects' where each had a dictionary of 'conditions' and each of those had a list of multiple 'responses' (NB this isn't going to work for you right now):

```
resps = allData['jwp']['preAdaptation'] #retrieve the response list for this condition
resps.append('correct') # added to the original but we don't have to write it all out
```

but occasionally you do want a copy though, because you *don't* want your changes reflected back in the original. For that we need to do a little more work:

```
import copy
stim4 = copy.copy(stim1) #the copy function in the copy module
stim4['word']='rat'
print stim1, stim4
```

2.4 Loops

Repeating things is what computers are good at and humans find boring! Repetition is controlled with loops, which come in two varieties. *for... loops* are when you want something to repeat a known number of times, whereas *while... loops* will run for an unspecified duration until some condition is met.

2.4.1 for... loops

If you've come from matlab programming you probably expect a for...loop to operate over some numbers, which you'll use to index some other object (a string or an array). In Python anything that knows how to 'iterate' can be used as the basis for a loop and will return its next value each pass through the loop.

For instance, strings and lists both know how to 'iterate'. As with *ifStatements* the contents of the loop is determined by the indentation:

```
for thisLetter in 'hello':
    print thisLetter
    print 'printed'
```

```
print 'loop done'

for thisWord in ("Hello World, I greet you").split()
    print thisWord

for thisInt in [1,2,3]:
    print thisInt, thisInt*3
```

Let's use a loop to create a list of dictionaries:

```
myList=[]
for thisInt in range(5):
    thisEntry = {}
    thisEntry['val']=thisInt
    thisEntry['X3']=thisInt*3
    myList.append(thisEntry)
print 'myList is now', myList

print 'printing one entry per line:'
for thisEntry in myList:
    print thisEntry
```

Jon's style tip: I quite often use 'this' as a way to remind myself that a variable was being used in a loop. It can lead to confusing bugs if you refer to a variable after a loop has ended that was designed for use in the loop. After the loop ends the variable still exists, but has the last value it was left with in the loop:

```
word = 'blah' #maybe this was defined earlier in the script

wordSet =[]
for word in ('Hello people').split():
        wordSet.append( {'word':word, 'upper':word.upper()} )

print word #I was still intending to print 'blah'
```

If the value that is returned during the iteration can be 'unpacked' further then it can be assigned to multiple values in the loop. For example:

```
a_b = [2,8]
a, b = a_b #'unpack' the variable a_b to a pair of variables
a, b, c, = a_b #error?
a, b = [1,2,3] #error?
```

If you use a dictionary in a loop it will return each of the keys:

```
man = {'name':'Jon', 'style':'geek', 'age':21}
for thisKey in man:
    print "This man's %s is %s" %(thisKey, man[thisKey])
```

Dictionaries also have an *items* method, which returns a list of key/value pairs. We could iterate over the list of pairs, which means we could do this:

```
for thisKey, thisVal in man.items():
    print "This man's %s is %s" %(thisKey, thisVal)
```

2.4.2 Nesting loops

You can nest one loop inside another (as deeply as you like). The inner loop will perform a full cycle on each pass through the outer loop:

```
for thisNum in range(5):
    for thisChar in 'abc':
        print thisNum, thisChar
```

Switch the order of the two loops and try it again.

Remember indentation is key in deciding which of the loops a code line belongs to:

```
for thisNum in range(5):
    print '------------starting run %i' %(thisNum)
    for thisChar in 'abc':
        print thisNum, thisChar
        print 'x'
    print '------------finished run %i' %(thisNum)
```

2.4.3 Enumerate

Often you'll want to know not only the current value in a list, but also its location. For instance, if we run some trials like this (NB. I'd definitely recommend doing this in the code editor rather than the Shell panel):

```
from numpy import random #we need a new lib for this demo
oris = [0,45,90,180]
resps = []
trials=[]
RTs = []
for thisRep in range(5):#repeat 5 times
    random.shuffle(oris) #NB this shuffles the list 'in-place'
    for thisOri in oris:

        #imagine we presented a stimulus

        #and simulate getting a response
        resp = round(random.rand()) #we'll create a random 'response'
        resps.append(resp) #the response from this trial
        RT = random.rand() #some number between 0-1
        RTs.append(RT)
        trials.append(thisOri) #also store what this trial was

#... later we want to print out what happened
for thisResp in resps:
    print thisResp
```

For the last part we could avoid looping through the *values* of `resps` and instead loop through a set of *indices* to fetch the values:

```
for ii in range(len(resps)):
    thisResp = resps[ii]
    thisTrial = trials[ii]
    thisRT = RTs[ii]
    print ii, thisResp, thisTrial
```

The need to know the current value AND it's index in the list is so common that Python has a special function for it built-in called `enumerate`:

```
for ii, thisResp in enumerate(resps):
    thisTrial = trials[ii]
    thisRT = RTs[ii]
    print ii, thisResp, trials[ii]
```

2.4.4 while... loops

If you want your loop to end based on some condition, rather than based on a certain number or iterations, then you could use a while...loop. For instance, an experiment might be based something on time rather than on repeats:

```
import time #time module is built into Python
t0=time.time() #time in secs
nReps = 0
while (time.time()-t0) < 0.5: #continue this loop for 0.5s
    n = n+1 # (or you could use the shorthand n+=1. Try that in the shell)
print 'we did %i loops in 0.5s' %(n)
```

Or you might want to end the loop only when a valid response has occurred.:

```
from numpy import random
validKeys = 'az'
availableKeys = 'azqwertyuiop'
resp=None #None is a special value in Python for, well, none!
while resp==None:
    ii = random.randint(0,len(availableKeys))
    keyPress = availableKeys[ii]
    if keyPress in validKeys:
        resp=keyPress
        print 'At last'
    else:
        print "'%s' was not a valid key" %(keyPress)
print "subject responded with '%s'" %(resp)
```

Other than that, while...loops are really similar to for...loops (personally I use them less).

2.4.5 *break* and *continue*

Sometimes you need to end a loop, or this repeat of a loop, prematurely. `break` allows you to end a loop completely and move to the next code after it. `continue` means 'continue to the next iteration of the loop without finishing this one'. They both only operate on the innermost level if your loops are nested.

Let's combine some of the earlier code. We'll run trials as in the *enumerate* demo and collect keypresses a bit like the *while... loops* section. But instead of waiting for a valid response, we'll just ignore trials where subjects responded got the wrong keys. And if they hit 'q' we'll abort the experiment:

```
from numpy import random #we need a new lib for this demo
validKeys = 'az'
availableKeys = 'azqwertyuiop'
oris = [0,45,90,180]
resps = []
trials=[]
RTs = []
for thisRep in range(5):#repeat 5 times
    random.shuffle(oris) #NB this shuffles the list 'in-place'
    for thisOri in oris:

        #imagine we presented a stimulus

        #now simulate getting a response
        ii = random.randint(0,len(availableKeys))
        keyPress = availableKeys[ii]
        RT = random.rand() #some number between 0-1
        #perform analysis
```

```
if keyPress == 'q':
    print 'experiment aborted'
    break
elif keyPress not in allowedKeys:
    print 'invalid response'
    continue # to next trial (don't analyse further)
#we got a useful trial so store info
resps.append(keyPress) #the response from this trial
RTs.append(RT)
trials.append(thisOri) #also store what this trial was
```

2.5 Conditionals

We need to be able to control what parts of a script get run based on *conditions*. For example, if this trial requires a probe to be presented then run *this code* but if not run *that code*.

2.5.1 Boolean logic

Python, like most programming languages represents things as being `True` or `False` and these correspond to 1 or 0. 'Comparison operators' (<, >, ==, <=, >=', !=) will return `True` or `False`. See what these return (some are obvious, some not):

```
a = 4
4<5
a>5
a==4
4=4 #error?
4.0==4 #compare a float with an int
a>=3
a=>3
a!=3
True==1
False==0.0
```

Note: In nearly all programming languages = means 'make this equal to' whereas == means 'test whether this is equal to'. But even experienced programmers will occasionally use the wrong one.

Conveniently Python does the 'right thing' when comparing strings using the same syntax:

```
a = "spam"
b='blah'
b > 'aaah'
b<'aaah'
b=='a'
b[2]=='a'
b.endswith('h')
'aaa'.endswith(a) #work out why this is wrong!
```

As well as those standard comparison operators, Python defines the operation `in` for testing the contents of any object that Python considers *iterable*:

```
'a' in 'blah'
10 in range(5) #maybe find out what range(5) does?!
```

```
subj = {'name':'jwp','age':21}
'age' in subj.keys()
5 in 10
```

As with any boolean logic system you can perform AND, OR, NOT operations on these things. These are usually typed with lower case (`and`, `or`, `not`):

```
a=='spam'
1==2 or a==4
1==1 and not a<4
subj['name'].startswith('j') and subj['age']<18
not True == False #get your head around this one!
5 not in [1,2,4]
```

Note: You can also use `&`, `|`, `!` for `and`, `or`, `not` but few people do. The text version is just more natural to read. You do sometimes see `!=` though.

As soon as one of the pieces of a boolean statement is `False` the rest of the statement is not evaluated. That's very important for some statements where you first need to check if it's possible to run the next function, or where evaluating one part will take processing time and isn't always necessary:

```
subj['weight']==83 and subj.has_key('weight')
subj.has_key('weight') and subj['weight']==83 #reorder
keys = ['y']
keys[0]=='y' #check the first key press
keys = []
keys[0]=='y'
len(keys)>1 and keys[0]
```

2.5.2 if... statements

If statements allow you to say, "if this statement evaluates to be `True` then run the next lines of code". To determine what count as the 'next lines' you have to indent the code. These examples are getting longer - you might want to switch to using the script editor panel if you've been using the shell so far:

```
if True:
    print 'hello'
    print "toast"
if  5==4:
    print 'no'
```

You can optionally define one or more `elif` statements and an `else` statement:

```
if 5==4:
    print 'crazy'
elif 3*5>10 and 'a'<'b':
    print 'possibly'
else:
    print 'catch all'
```

If I've used an if...elif... I find it's often a good idea to include an `else` statement even if you don't think it will be called. It can make it easier to find bugs later on if something surprising happens:

```
resp = ['left']
if resp=='left':
    corr=True
```

```
elif resp=='right':
    corr=False
else:
    print "Response should be 'left' or 'right' not %s" %(resp)
```

Note: The fact that Python will interpret either " or ' as a string makes it very easy if you want a string to contain one of those characters. e.g. "Won't hurt" is fine but 'Won't hurt' will cause an error (because the string effectively ends after the *n*). If you want to be really safe you can start or end a string with triple quotes and then the string can contain either type of quote inside.

Statements can nest too. Make sure you understand whether each of the following lines will be run and why:

```
resps = [0,1,1,0]
if len(resps)>0: #subj responded
    print "mean resp=", sum(resps)/float(len(resps))
    if resp[0]==0:
        print 'first resp correct'
    else:
        print 'first resp incorrect'
    print 'hello'
print 'done'
```

2.6 Functions

You've learned all the key elements to programming. Variable, conditionals and loops allow you to do pretty much anything your computer can do. But your code will quickly become hard to read if you simply write everything line after line and copy-and-paste pieces of code you want to re-use.

Functions allow you to a) reuse your code and b) make it easier to read by hiding away the guts of your code. For instance, if we have a pair of points in space, pt1 and pt2, with (X,Y) coordinates. We could find the distance between those using Pythagoras' theorem:

```
pt1 = [8.0, 5.3]
pt2 = [2.1, 9.2]
sep1 = ((pt1[0]-pt2[0])**2+(pt1[1]-pt2[1])**2)**0.5
```

Now, the problem is that a) each time you see that in your code you'll have to spend a moment working out why it's there, and b) each time you write it you might make a mistake. Let's re-use the code for two more variables later in our script.:

```
pt3 = [3.0, 8.1]
pt4 = [2.5, 4.2]
sep2 = ((pt3[0]-pt4[0])**2+(pt1[1]-pt4[1])**2)**0.5
```

Did you notice that when I changed the pt1,pt2 values to pt3,pt4 I missed one? Hard bugs to spot, right? OK, the solution is to write functions to do these simple jobs and hide away the hard-to-read pieces of code so that the rest is clear. In this case we want a function that takes two values and finds their separation. Type this function into the shell (press and extra <*return*> when you're done to dedent (ends the definition of the function):

```
def separation(a, b):
    """Get the distance between two points"""
    sep = ((a[0]-b[0])**2-(a[1]-b[1])**2)**0.5
    return sep
```

Having defined that function you can use it repeatedly:

```
sep1 = separation(pt1, pt2)
sep2 = separation(pt3, pt4)
```

Did you notice how help was provided for the function when you started the brackets? It was based on the *"""textHere"""* in your *separation* function. In Python these are called docstrings.

Note: Almost any time you're planning to do something multiple times in a script you should think about replacing it with a function.

2.6.1 Function inputs

On the whole you should assume that your function only knows about the things that it was given in the first line (you can also have *global* variables that can be seen from anywhere, but these are generally to be avoided). In the function above we required two variables and these were given the names *a* and *b* as we received them (it doesn't matter what names they had in the script).

Python allows you to specify the inputs to a function by order, or by name, or by a combination. e.g. we could have called the function like this:

```
sep1 = separation(a=pt1, b=pt2)
```

You can also provide default values for inputs so that these don't have to be specified each time. For instance:

```
def separation(a, b=[0,0]):
    """Get the distance between two points, or from the origin for a single point"""
    sep = ((a[0]-b[0])**2-(a[1]-b[1])**2)**0.5
    return sep

sep3 = separation(pt4) #b is set to [0,0]
```

Because we can use names for the arguments, we don't have to specify all those that precede the one we care about. e.g.:

```
def separation(a, b=[0,0], verbose=False):
    """Get the distance between two points, or from the origin for a single point"""
    if verbose:
        print 'vertical sep=', a[1]-b[1]
        print 'horizontal sep=', a[0]-b[0]
    sep = ((a[0]-b[0])**2-(a[1]-b[1])**2)**0.5
    return sep

sep4 = separation(a=pt3, verbose=True) #use first and last args, middle is default
sep5 = separation(pt4, pt2, verbose=True) #can combine ordered args and names
sep6 = separation(a=pt4, pt2) #error? after a named arg, all others must be named
```

2.6.2 Function outputs

Some functions don't need to return anything (they just perform an operation like present a stimulus). The function above returns a single value. You can return multiple values too. Or you can choose whether to return one or more values in the function (this might not be wise though!):

```
def separation(a, b=[0,0], verbose=False):
    """Get the distance between two points, or from the origin for a single point"""
    vert = a[1]-b[1]
    horiz = a[0]-b[0]
```

```
    sep = (vert**2 + horiz**2)**0.5
    if verbose:
        print 'vertical sep=', vert
        print 'horizontal sep=', horiz
        return sep, horiz, vert
    else:
        return sep

sep5, horiz5, vert5 = separation(pt4, pt2, verbose=True)
#or store them as a tuple of 3 values:
sepInfo5 = separation(pt4, pt2, verbose=True)
print sepInfo
```

Note: If you don't specify any return values in your function but you then try to assign a variable to the output, then that variable will just become equal to None (i.e. it doesn't automatically raise an error).

2.6.3 Operations in-place

Remember that Python passes pointers to objects rather than copies of them, unless you manually make a copy. So again, if you do anything that changes the variable 'in-place' then it will be changed in the code that called your function.:

```
def changeStr(input):
    input[0] = 'a'

myStr = 'baah'
new = changeStr(myStr)
print new #surprise? (see note above)
```

2.7 Modules

You can put functions that you use repeatedly into a file so that you don't end up rewriting them at the top of your script. Save your *separation* function into a file called tools.py. Now, in the same directory, create a new file (e.g. 'importingExercise.py'):

```
import tools
print tools.separation([1,2],[6,4])
```

You could also do:

```
from tools import separation
print separation([1,2],[6,4]) #now we don't need tools._____
from tools import * #Not recommended - see below
```

Note: It's tempting to import everything from a module into the main *namespace* so that you don't have to keep typing module._____ . In fact it isn't a good idea because if you have many things defined in your modules you can find that you've overwritten one of your functions with a variable (or vice versa). Again it creates bugs that can be really hard to find.

And you can even rename things as you import them. *numpy* is a common library for numerical operations (as we've seen) and most people import that like this so that it only takes 2 characters to call the functions:

```
import numpy as np
print np.ones([2,3])
```

So that syntax of a dot is used in various ways in Python:

```
import numpy, os
os.getcwd() #the getcwd function in os module
numpy.random.rand() #the rand function in the random submodule of numpy
("hello").upper() #the upper method of a string object
```

You can also have multiple modules within a folder. Then you may also need to add a file called __init__.py which can, optionally, run some code every time you import things from this folder. e.g. I might have a folder in my home directory like:

```
HOME/
    python/
        jwpTools/
            __init__.py
            geometry.py
            filters.py
            sounds.py
```

If I add jwpTools to my path (see below) then I can do this in any Python script I run:

```
from jwpTools import sounds, filters
import jwpTools.geometry as geom
```

2.7.1 Adding a location to your path

In a 'normal' python installation you could add HOME/python to your path by finding the site-packages directory of your installation and adding a text file there called *anythingYouLike.pth* and containing any paths that you want on separate lines (e.g. HOME/python in our case). If you use the PsychoPy Standalone installation then specify your folder in the preferences>general>paths (e.g. ["HOME/python"]) and then you need to make the one extra step of importing the psychopy lib before importing your own libs:

```
import psychopy #importing this causes the addition of paths in prefs
import jwpTools.sounds
```

2.8 Classes

You've possibly realised that everything in Python is an 'object'. Everything has data stored inside it and has methods that allow that data to be accessed or modified. For instance, a string is an object whose data are the values of the letters and whose methods are *upper()*, *lower()*, *split()*... It's very convenient that objects can have methods associated with themselves.

In PsychoPy you'll find objects like windows and stimuli with methods like *draw()*, *setPos()*, *setImage()*.

Classes aren't essential for most programming of experiments, but if you're comfortable with this object-oriented concept of programming, and comfortable with how to create functions already then classes provide you with a way to create your own custom objects, or to modify existing classes of object. (If you aren't yet comfortable with function definitions then leave this section until another day!)

A class is just a collection of functions and variables to store/retrieve/modify related data. Every class has an __init__ function to be run when an 'instance' of that class is created (*'a'* is an instance of a *string*).

Note: In Python functions that start with _ indicate internal functions/attributes that the user should probably ignore. Methods start/end with __ indicate functions that typically have special meanings to the language. For instance *__init__* is a function that is run when an object is created, *__str__* is a function that is run if an object is being converted to a string representation etc.

Let's define a simple class to represent a moving dot (we'll stick to one axis of motion for simplicity). To know where a dot is at any point in time we would need to know its start position and rate of motion. We'll give our class two methods, one to initialise it and another to report the position for a point in time. I usually give class names a capital letter so I remember what contents of my modules are classes and what are simple functions:

```
class Dot:
    def __init__(self, startPos, speed):
        self.startPos = startPos
        self.speed = speed
    def getPos(self, t):
        """Get the location of the dot at a given time"""
        pos = self.startPos + t*self.speed
        return pos

dot1 = Dot(2.0, 1.5) #this is really just a call to the Dot.__init__() function
print dot1.getPos(3.0) #after 3 secs this dot should be at pos=6.5
print dot1.speed #the value of self.speed can be accessed like this
dot2 = Dot(2.0, -1.5) #each instance of the Dot is independent
print dot2.getPos(3.0)
print dir(dot2) #what attributes does a Dot instance have?
```

Look carefully at that code. For every function within the class there's an extra input argument, which almost every programmer in the world will call *self*. So a class method with 3 arguments should be called with 2 (*self* is going to be given automatically). *self* allows you to set/retrieve variables that are associated with your object.

It might be useful to have the dot keep track of time. Then we could find its current pos without having to tell it a time. Or we could use a built-in clock unless the user requests a specific time:

```
import time #the time module built in to Python

class Dot:
    def __init__(self, startPos, speed):
        self.startPos = startPos
        self.speed = speed
        self.t0 = time.time() #the current time
    def getPos(self, t=None):
        """Get the location of the dot at the current time (or
        any time, if given)"""
        if t == None:#if we were given a value for t use that, otherwise...
            t = time.time()-self.t0 #the time since t0
        pos = self.startPos + t*self.speed
        return pos

dot1 = Dot(2.0, 1.5)
time.sleep(1.5) #pause the script for 1.5 secs
print dot1.getPos() #with no args get the current pos
print dot1.getPos(5.0) #pos at some given point in future
```

2.8.1 Sub-classing

One big advantage of classes is that you can take an existing class and modify just one part of it. This code creates a subclass of string that has all the methods of normal strings but adds the additional method of returning itself as a

sentence(). All other methods (including *__init__*) are untouched.:

```
class NewStr(str):
    def sentence(self):
        #capitalise the first character
        self = self.replace(self[0], self[0].upper())
        #add a full stop
        if not self.endswith('.'):
            self = self+'.'
        return self

x = NewStr('hello everyone')
print x.sentence()
print dir(x) #compare this with dir of a normal string
```

You can also replace existing methods in exactly the same way.

2.9 Exercises

2.9.1 Exercise 1.1 - using loops

Write a script that generates this output (can be done in 2 lines):

```
1 4
2 8
3 12
4 16
5 20
```

Go to *Solution 1.1*

2.9.2 Exercise 1.2

Write a script that generates this output (can be done in 4 lines):

```
0 0
hello
1 4
hello
2 8
hello
3 12
hello
4 16
hello
______done
```

Go to *Solution 1.2*

2.9.3 Exercise 1.3

Write a script that generates this output (can be done in 3 lines):

```
0 a
0 b
0 c
1 a
1 b
1 c
2 a
2 b
2 c
3 a
3 b
3 c
4 a
4 b
4 c
```

Go to *Solution 1.3*

2.9.4 Exercise 1.4 - make a dict

Create a dictionary with your name, age, house number and street

Go to *Solution 1.4*

2.9.5 Exercise 1.5 - loop over a dict

Take your dictionary from *Exercise 1.4 - make a dict* and print off each key and value, to make an output formatted like this:

```
house: 33
age: 21
street: Banana Drive
name: jon
```

Go to *Solution 1.5*

2.9.6 Exercise 1.6 - a list of dicts

a)

We want to run the Stroop task. To do this we need a set trials (I would use a list of dictionaries) that have keys:

- 'word' (one of 'red','green','blue')
- 'ink' (one of 'red', 'green','blue')

We want every ink colour to be paired with every word. This can be done with 4 lines of code (but 5 is reasonable).

b)

In a loop print step through the entries of your trials and print out something like this:

```
red red
red green
red blue
...
blue blue
```

c)

Print out just the ink colour of just the 4th trial in your list. If it didn't match the 4th row of

d)

Repeat Part b) but add the index of each trial to your printout:

```
0 red red
1 red green
...
7 blue green
8 blue blue
```

Go to *Solution 1.6*

2.9.7 Exercise 1.7

You can actually do *Exercise 1.6 - a list of dicts* as a dict of lists instead of a list of dicts. When you do that, printing the output of trials will give:

```
{'word': ['red', 'red', 'red', 'green', 'green', 'green', 'blue', 'blue', 'blue'], 'ink': ['red', 'gr
```

Go to *Solution 1.7*

2.9.8 Exercise 1.8 - formatted strings

Given these variables only:

```
a = 'Hello'
name = 'Jon'
height = 1.80
hour = 9
minute = 5
```

use string formatting to print out the following:

> Hello Jon HELLO Jon The time is 9.05am The time is 09:05 Jon is 1.8m tall

Go to *Solution 1.8*

2.9.9 Exercise 1.9 - explore modules

From the *sys* module find out the:

- the current version of Python
- the name of the platform you're running on

- the default encoding (ASCII or Unicode)
- the location of the Python executable currently running

From the *os* module find the:

- current working directory
- the current environment variables (paths , proxies etc.) for your operating system

Work out from *numpy.random* how to generate:

- a set of random numbers with shape = [2,3] (NB `rand()` won't work for this)
- a random *integer* between 5 and 10

Go to *Solution 1.9*

2.9.10 Exercise 1.10 - create an abs() function

Let's create a function that returns the absolute value of a numeric (a float or int). Actually Python includes one called `abs()`, but we'll make our own called `absol()` for sake of the exercise.

1.10 a)

Use an *if* statement that will reverse the sign of a variable *x* if *x* is negative (that's all an abs() function does).

1.10 b)

Insert your if statement into a function called `absol` so that it returns the correct value.

1.10 c)

Test the output of your new function against the built-in `abs()` for 100 random numbers between -1 and 1. Count the number of fails and report with a `print` statement at the end whether the method works.

Go to *sol1.10*

2.10 Solutions

2.10.1 Solution 1.1

```
for n in range(5):
    print n, n*4
```

2.10.2 Solution 1.2

```
for n in range(5):
    print n, n*4
    print 'hello'
print '______done'
```

2.10.3 Solution 1.3

```
for n in range(5):
    for c in 'abc':
        print n, c
```

2.10.4 Solution 1.4

```
me = {'name':'Jon','age':21,'house':33,'street':'Banana Drive'}

#or:
me={}
me['name'] = 'jon'
me['age'] = 21
me['house'] = 33
me['street'] = 'Banana Drive'
```

2.10.5 Solution 1.5

```
me = {'name':'Jon','age':21,'house':33,'street':'Banana Drive'}
for key in me:
    print "%s: %s" %(key, me[key])

#the last line could be
print key, ":", me[key]
```

2.10.6 Solution 1.6

```
#part a)
trials=[]
for word in ['red','green','blue']:
    for ink in ['red','green','blue']:
        thisTrial = {'word':word, 'ink':ink}
        trials.append( thisTrial )

#part b)
for thisTrial in trials:
    print thisTrial['word'], thisTrial['ink']

#part c)
print trials[3]['ink'] #4th entry has index 3 BECAUSE WE START AT zero

#part d)
for ii, thisTrial in enumerate(trials):
    print ii, thisTrial['word'], thisTrial['ink']
```

2.10.7 Solution 1.7

```
trials={'word':[], 'ink':[]}
for word in ['red','green','blue']:
    for ink in ['red','green','blue']:
```

```
        trials['word'].append(word)
        trials['ink'].append(ink)
print trials
```

2.10.8 Solution 1.8

```
greet = 'Hello'
name = 'Jon'
height = 1.80
hour = 9
minute = 5

print "%s %s" %(greet, name)
print "%s %s" %(greet.upper(), name)
print "The time is %i.%02iam" %(hour, minute)
print "The time is %02i:%02i" %(hour, minute)
print "%s is %.1fm tall" %(name, height)
```

2.10.9 Solution 1.9

```
import sys
print sys.version
print sys.platform
print sys.getdefaultencoding() #NB this is a function not a variable
print sys.executable
import os
print os.getcwd()
print os.environ
```

Note: Python built-in modules tend to use the word get___() to indicate that this is a function rather than an attribute. That might be a good thing to do in your own functions?

```
import numpy as np
print np.random.random([2,3])
print np.random.randint(5,11) #NB the high-end of the interval is non-inclusive :-/
#or
print round(np.random.random()*5+5)
```

CHAPTER

THREE

DAY 2 - PSYCHOPY PROGRAMMING

General issues (on the PsychoPy.org documentation):

CHAPTER

FOUR

DAY 3 - ANALYSING DATA IN PYTHON

4.1 Numpy (numerical python)

TODO

4.2 Matplotlib - Matlab-like plotting library

TODO

4.3 Importing PsychoPy data files

TODO