Introduction

These are getting started notes on Python. I offer an error bounty of between 20p and 2 pounds for mistakes. Contact me at `conor.houghton@bristol.ac.uk` or come up after a lecture.

Computing in Computational Neuroscience

People in computational neuroscience usually use either MATLAB or Python or they use a specialized simulation tool such as GENESIS or NEURON or NEST. These simulation tools are optimized to efficiently simulate large complicated neurons, in the case of GENESIS or NEURON, or last and complicated networks of neurons, in the case of NEST. The simulation tools have been writen over many years and can be quite complicated and idiosyncratic, however, NEURON at least, now has a Python interface.

MATLAB has some advantages, it has a very large user base across many parts of applied mathematics, a huge collection of libraries and package. There is a free community authored version called Octave, they are not completely compatible and Octave lacks many libraries and features. Matlab uses a matrix paradigm which is easy to use and quick when you get used to it. Python is a proper programming language, with a nicer language structure than Matlab, it is free and open and has many libraries, though perhaps not as many as Matlab.

These notes are only the roughest of introductions to Python, remember google and stackoverflow are your friends.

Python - getting started

Python either runs as a script or on an interpreter. To get the interpreter you type `python` on the command line and you get something that looks like this:

```
Python 2.7.5+ (default, Sep 19 2013, 13:48:49)
[GCC 4.8.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>>
>>>
```

where the numbers on the right are just for these notes.

You can use python on the interpreter as a glorified calculator

```
>>> 5+6
11
>>> 12/9.0
1.3333333333333333
>>>
```

It has big numbers which can be useful sometimes

```
>>> 2**200
1606938044258990275541962092341162602522202993782792835301376L
```

The `L` at the end is because it is a long number. To get special functions and so forth you need to `import` the math package

```
>>> import math
>>> math.tan(math.pi/4.0)
0.9999999999999999
```

The syntax for the python math package is more or less the same as for `math.h` in C or `cmath` in C++. Notice the namespacing, you need to write `math.tan(math.pi/4.0)`. If you want to import the functions and so on into your namespace you use `from`, eg

```
>>> from math import tan
>>> from math import pi
>>> tan(pi/4.0)
0.9999999999999999
```

and, if you want to import everything from `math` into your current namespace it is

```
>>> from math import *
>>> sin(pi)
1.2246467991473532e-16
```

Of course, we don't want to use it as an interpreter for anything very complicated, so we write scripts. However, as an interpreted language you don't compile it, so if you have a program called `foo.py` you write

```
> python foo.py
```

to run it, here the '¿' denotes the command prompt. Alternatively, you can add a shebang to `foo.py`, so in my case I would add #!`/usr/bin/python` as the first line of `foo.py` and change `foo.py` to executable by typing `chmod u+x foo.py` and then I can run `foo.py` directly

```
> ./foo.py
```

Of course, you might need to change `/usr/bin/python` to something else on your machine, it depends on the output of `which python`.

Python - some basics

There are lots of introductions to Python on the web and that's probably the best place to look. Variables are not declared

```
hello = "hello world"
print hello
```

prints hello. It has a nice slice notation

```
hello = "hello world"
print hello[0]
print hello[1:4]
print hello[-1]
```

prints `h`, `ell` and `d`, `hello[-1]` gives the last character.

One of the most distinctive features is that blocks are created by indenting rather than brackets. In

```
hello = "hello world"

```

```
for a in hello:
    print a,
print
```

the block for the for loop is `print a,` because of the indent, not the colon after the `for` statement too. This sort of for loop, looping over parts of an object, is considered more pythonic than the indexed loops in C++ and so on, this is possible though

```
hello = "hello world"

for i in range(0,len(hello)):
    print hello[i],
print
```

If you do want the index and the object it refers to you should use an enumeration:

```
hello = "hello world"

for i,a in enumerate(hello):
    print i,a
print
```

The if statement is also indented, well all blocks are, but for completeness

```
import random

a=random.randint(1,3)

if a==1:
    print 'one'
elif a==2:
    print 'two'
else:
    print 'three'
```

For some reason what are called arrays or vectors in other languages are called lists in python

```
hello = "hello world"

letters = []

for a in hello:
    letters.append(a)

print letters
```

Lists need not be heterogenous

```
list=[72,79,86,96,103,"Catheral Parkway"]
print list
```

Python also has a tuple type, like a list but immutable.

One thing about Python that is hard to get used to if you are used to C or C++ is that what I keep calling variables are more correctly labels, they name memory locations rather than pieces of data. This often catches me out, to see the difference look at

```
a = [0,1,2]
b = a

print a
print b

b[1]=-1

print a
print b
```

and note that changing `b[1]` has changed the value of `a[1]`.

Python - functional features

Python has some stylish functional programming feactures for working with lists, for example `map` does a function on all the elements in a sequence so

```
def cube(x):
    return x*x*x

numbers = [1,2,3,4,5]

cubes = map (cube,numbers)

print cubes
```

where you should also note the syntax for defining functions, there are functions and classes in the usual way, though classes have no private members. In fact this code can be made more streamlined, there is a construction for avoiding giving names to things that don't need to be named:

```
numbers = [1,2,3,4,5]

cubes = map (lambda x:x*x*x,numbers)

print cubes
```

In fact, there is also a list comprehension type construction that also does the same thing

```
numbers = [1,2,3,4,5]

cubes = [x*x*x for x in numbers]

print cubes
```

Python contains a number of these functional commands in addition to `map` and a number of different data structures, part of the skill of writing 'pyhtonic' code is to make these work for you.

Scientific calculation packages

Python has a huge number of packages. For an introduction to some of them in the context of a very annoy / fun puzzle try `http://www.pythonchallenge.com/`. Here we will quickly introduce the packages commonly used in scientific computing

- *Numpy* This adds matrix and vector datatypes and enables fast vectorized calculations, it also includes methods for finding eigenvectors and eigenvalues.
- *Scipy* This adds various numerical routines for working out things like integrals and the solution to differential equations.
- *matplotlib* Plots stuff.

Here `numpy` is imported with the name `np`

```
import numpy as np
d1=np.array([1,-1,1])
d2=np.array([1,2,1])
print d1
print d2
print d1*d2
print np.dot(d1,d2)
```

so we define two numpy arrays, we see that `*` gives element-by-element multiplication, whereas `np.dot(d1,d2)` gives the dot product. We can do matrix multiplication as well

```
import numpy as np
d1=np.array([(1,-1,1),(1,2,1),(1,-1,1)])
print d1
d2=np.array([1,3,1])
print d2
print np.dot(d1,d2)
```

and lots of linear algebra

```
import numpy as np
d1=np.array([(1,-1,3),(1,2,1),(3,1,1)])
print np.linalg.det(d1)
```

As for `matplotlib` here is a simple example

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10)
plt.plot(x, np.sin(x), linewidth=2)

plt.savefig("example.png")

plt.show()
```

It both saves the plot as `example.png` and shows it on the screen.