Computing

These are getting started notes on Python, the name of the corresponding example program is given in brackets in a fixed width font (`fixed width font`) just before the code listing.

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 large 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.

A few years ago MATLAB had some advantages, until recently it had a very large user base across many parts of applied mathematics, a huge collection of libraries and package. 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 for some specialist applications, like wrangling EEG data, but more for machine learning applications. For this reason, MATLAB is quickly disappearing and the field is moving to the jupyter languages: Python, Julia, and R.

Other commonly used tools include Brian, a Python based package which supplies efficient specification and integration of differential equations used in neuronal simulations, XPP which is useful for analysing the dynamics of the differential equations used in neuroscience, NeuroML which is a meta-language for describing neuronal models and R which is used for statistics.

These notes are only the roughest of introductions to Python; Google and StackOverflow are your friends. Conor Houghton has a website with some hopefully fun coding puzzles if you'd like to try it:

- `www.choicetask.com/python`

If you do try it, please forward any feedback and suggestions for new pages to `conor.houghton@gmail.com`.

Python - getting started

Python either runs as a script or on an interpreter. To get the interpreter you type `python` or `python3` on the command line and you get something that looks like this:

```
Python 3.10.12 (main, Jun 11 2023, 05:26:28) ...
Type "help", "copyright", "credits" or "license" ...
>>>
>>>
```

where the numbers on the right are just for these notes. The '3.10.12' is the version number. Up until April 20, 2020, two incompatible versions of Python: Python 2 and Python 3, were still in use. Since April 20, 2020, Python 2 has ceased development and all code you will use in this course will be in Python 3.

You can use python on the interpreter as a glorified calculator

```
>>> 5+6
11
>>> 12/9
1.3333333333333333
>>>
```

It has big numbers, which can be useful sometimes

```
>>> 2**100
1267650600228229401496703205376
```

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 there is a namespace, 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 `#!` (pronounced "shebang") to `foo.py`. In my case I would add `#!/usr/bin/env python3` 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
```

The initial `#!/usr/bin/env` uses the `env` command to find the appropriate python executable on your system, and this command should exist on POSIX compliant environments (MacOS and most Linux distributions). You can view the location found by `env` with the shell command `which python3`. You may need to change these paths to point to the python executable on your machine using this output of `which python` or `which python3`. If you use the command `python`, verify that it points to `python3` rather than an older installation of`python2`. If you are using a different setup, you may need to specify the full path to your python3 interpreter executable in the `#!` line.

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_world.py`)

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

prints hello. It has a nice slice notation (`hello_world_slice.py`)

```
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_world_for.py`)

```
hello = "hello world"

for a in hello:
    print(a,end='')
print()
```

the block for the for loop is `print(a)`, because of the indent, note 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_world_indexed.py`)

```
hello = "hello world"

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

If you do want the index and the object it refers to you should use an enumeration
(`hello_world_enumeration.py`):

```
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 (`if.py`)

```
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 (`if.py`)

```
hello = "hello world"

letters = []

for a in hello:
    letters.append(a)

print(letters)
```

Lists need not be heterogenous (`hetro_list.py`)

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

Python also has a tuple type, like a list but immutable. The `enumerate` we saw above makes an `enumerate` object, but you can cast it to a list of tuples (`enumerate_object.py`):

```
hello = "hello world"
print(enumerate(hello))
print(list(enumerate(hello)))
```

or

```
print([*enumerate(hello)])
```

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 sometimes behave as *references* or *pointers*. They name memory locations rather than pieces of data. This often catches me out, to see the difference look at (`labels.py`)

```
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]`.

To complicate matters, Python allows *shadowing* of variable names. Ra-assigning an existing symbol binds the symbol to a new memory location, leaving the old one untouched. For example, the following code

```
a=b=[1]
print(a,b)
a[0]=2
print(a,b)
a=[3]
print(a,b)
```

outputs

```
[1] [1]
[2] [2]
[3] [2]
```

In the above code, the symbols `a` and `b` are initially references to *the same* list, which contains a cell with the value 1. Changing the contents of this cell by assigning to `a[0]` also changes `b[0]`. However, when we create and bind the new list [3] to a, this new list now behaves separately from the one bound to b.

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 (map.py)

```
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 (map_lambda.py):

```
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.

The class construction has a few annoyances; for example each class can only have one constructor, which is defined using the `__init__` function (`class_example.py`).

```
class Counter:

    def  __init__(self,value):
        self.value=value

    def add(self,increment)
        self.value+=increment

counter=Counter(5)
print(counter.value)
counter.add(7)
print(counter.value)
counter.value+=3
print(counter.value)
```

Member variables have the `self` prefix and notice that if a class function uses a member variable then `self` has to be one of its arguments. Finally, as mentioned before the member variables are not private.

Scientific calculation packages

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

- *PyTorch* This is a machine learning library, but it also adds matrices and vectors. It is quickly taking over form Numpy.
- *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` (`np_array.py`)

```
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 (`np_matrix.py`)

```
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 (`np_det.py`)

```
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 (`plot.py`)

```
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.

Matplotlib also provides a module called *pylab* that is designed to resemble Matlab's API. You may see code that uses `from pylab import *` to import all functions from matplotlib.pyplot, numpy, numpy.fft, numpy.linalg, and numpy.random, and some other things. This shadows some builtin functions like `sum`, which can lead to strange bugs. Although usually discouraged, pylab can provide a simpler interface for interactive data exploration if you are aware of these issues.

Julia

Julia is a compiled programming language that runs much faster than Python; it is possible to write fast code in MATLAB where everything is written as linear algebra or using carefully optimized numpy or fancy just-in-time compilation in Python. The idea of Julia is that it runs at C-like speeds for natural, modern-looking code. It does this by allowing, while not requiring, type declaration and by not having classes, instead it has *types*, a bit like *structs* in C, and multiple dispatch.

It has other features to make it useful for scientific computing, little things like being able to `2v` when you mean `2*v`, along with big things, like a sophisticated multi-dimensional array datatype that can be used for efficient matrix operations. Presumably to help persuade people to finally abandon MATLAB it has a MATLAB-like syntax, blocks are denoated using a `end` keyword, the first element of an array `a` is `a[1]` and `1:10` means one to ten, not one to nine.

If you are used to Python, Julia can seem frustrating to debug, mostly because the typing can be hard to get used to, but debugging Julia as a Python programmer really reminds you how often you cast variables without even noticing; this is part of why Julia is much faster.

This only outlines the simplest parts of the language, the wikibook

> https://en.wikibooks.org/wiki/Introducing_Julia/

is a good place to look for a longer introduction. There is online Julia at

> https://juliabox.com/

A simple example

Here is a programme to add powers of two (`add.jl`):

```
1 highest_power=10
2
3 value=1.0::Float64
4 current=0.5::Float64
5
6 for i in 1:highest_power
7    value+=current
8    current*=0.5
9 end
10
11 println(value)
```

Line 1 defined `highest_power`; this is dynamically typed, as in Python, but `value` and `current` are given a type, `Float64`; as an indication of how seriously it takes typing, is **line 3** was `value=1::Float64` it would return an error since `1` isn't a `Float64`. You can find a full list of types in the wikibook, it has lots of different int and float types, along with rational numbers using `//` to seperate numerator and divisor (`rational.jl`):

```
a=2//3
b=1//2
println(a-b)
```

Arrays

Arrays are what Python calls lists, python is the odd one out here, array is a more common name. Julia has the same slicing functionality as Python, although as mentioned above indexing is different (`slice.jl`)

```
1 a=[1,2,3,4,5]
2 println(a[1:3])
3
4 for i in a
5    println(i)
6 end
```

prints `[1,2,3]` from **line 2** , **line 4** to **line 6** demonstrates a for loop. The last element in an array is indexed `end` so in the programme above `a[end]` is 5.

Arrays can store mixed items, but the array can be typed, so a in (typed_list.jl)

```
a=Int64[1,2,3,4,5]
push!(a,6)
println(a)
```

can only store items of type Int64. push! pushes an item onto the list, like append in Python, again, this is the more common notation. The ! is part of an convention where all commands that change an array have a !.

As mentioned above, Julia arrays can be multidimensional and have matrix like operations, but this won't be explored in this brief overview. There is also a tuple type which is immutable.

Functions

Here is a programme with some functions (functions.jl)

```
function add_to_int(a::Integer,b::Integer)
         println("int version")
         a+b
end

function add_to_int(a::Real,b::Real)
         println("float version")
         convert(Int64,a+b)
end

function add_to_int(a,b)
         println("what are these things")
         0
end

println(add_to_int(12,6))
println(add_to_int(12.0,6.0))
```

Obviously this is a very artificial example, but it shows some of the features of functions, first, their return value is the most recently evaluated expression and second, and more importantly, they support multiple dispach; the function is chosen to match the type of the arguments, here there is one function for `Integers`, this is a supertype which includes, for example, `Int64`, there is one for `Real`, the supertype that includes various floats, and one with no type; the correct function is used for each. If there is no correct fuction there will be an error.

There is also a terse 'mathematical functions' style function syntax that is useful, well, for functions that do the sort of things mathematical functions do (`math_fxn.jl`)

```

f(x,y)=2x+y

println(f(1,3))
```

You can also return more than one value (`multiple_return.jl`)

```
function powers(x)
         x,x^2,x^3
end

multiples(x::Float64) = x,2x,3x

a,b,c=powers(2)

println(a,' ',b,' ',c)

a,b,c=multiples(2)

println(a,' ',b,' ',c)
```

Composite Types

Julia doesn't have classes, this comes as a surprise at first, but it does have composite data types that work like structs, combining this with multiple dispach captures important parts of the functionality of classes, in a way that supports fast code (`struct_example.jl`

```
mutable struct Cow
        name::String
        age::Int64
```

```
end

mutable struct Poem
        name::String
end

function move(cow::Cow)
         print(cow.name," walks forward")
         println("showing the weight of her ",cow.age," years")
end

function move(poem::Poem)
         println(poem.name, " moves us to tears with its beauty")
end

poem = Poem("The Red Wheelbarrow")
cow = Cow("Hellcow",42)

move(cow)
move(poem)
```

You can see that although the structs have no methods, the function `move` can have different meaning for the two different data types. The default constructor defines the variables in the order they appear, it is possible to define other constructors, but that won't be considered here.

You can write constructors for these structs as ordinary functions; you can also use the key word `new` to write internal constructors, these serve, typically, to impose constraints on the input, see `constructor.jl`

```
struct Joke

    question::String
    answer::String

    function Joke(question::String,answer::String)
        if question[end]!="?"
            question=string(question,"?")
        end
```

```
        new(question,answer)
    end

end

function make_joke()
    Joke("what weapon does a fat jedi use","a heavy sabre")
end

joke=make_joke()

println(joke.question)
println(joke.answer)
```

Making functions

Functions are objects just like any other, so they can be returned like other objects (`make_function_1.jl`):

```
function make_adder(a::Int64)
    function adder(b::Int64)
        a+b
    end
end

three_adder=make_adder(3)
two_adder=make_adder(2)

println(two_adder(5)," ",three_adder(5))
```

or even (`make_function_2.jl`)::

```

function make_adder(a::Int64)
    b::Int64->a+b
end

three_adder=make_adder(3)
two_adder=make_adder(2)
```

```
println(two_adder(5)," ",three_adder(5))
```

Note: As of 2024 some application-specific machine-learning libraries for Python (e.g. Google's *Jax*) can just-in-time compile subsets of Python to run in parallel. Optimisations in these libraries for certain "tensor" computations (parallel operations over large multi-dimensional arrays) allow them to outperform Julia on certain tasks resembling e.g. deep learning, even when executed on the CPU rather than GPU. These libraries, however, come with their own restrictions and particularities. This landscape is constantly in flux and you will find that that various benchmarks change as hardware and software evolves.