Taken liberally from *Python for physical modeling* by JM Kinder and P Nelson

Getting help

In `IPython`, type

```
help(command)
```

to find information on `command`

Use the web, particularly

```
stackoverflow.com
```

Keep a log of commands and tricks that you find useful.

Assignment: defining variables

Assignment means to give a variable a new value:

```
a= 1
name= `Ste11'
print(a)
print(name)
a= 3
```

Python expressions: doing calculations

Numbers

```
1.2
1.0e6
```

Arithmetic

```
2**2 - 4
3*a - b
(3*a - b + c)/2
```

Expressions in parentheses are evaluated first.

Specialising Python

To analyse data, we need to augment Python's abilities by importing extra commands. Typing

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

gives new commands for numerical calculations and statistics and for plotting data. For example,

```
np.sqrt(2)
np.log(3.4)
```

Note we have to prefix with `np` to access this new commands.

Objects

Everything in Python is an object and can have associated attributes (specialized data) and methods (specialized functions)

Integer objects

whole numbers

```
i= 1
int(1.2)
```

Float objects

floating point numbers

```
a= 1
a= 1.0
a= 4/3*1.0e4
a.is_integer()
float(i)
```

a method (pointing to `a.is_integer()`)

For labels and text use string objects

```
s= “Ste11”
s= ‘Ste11’
s= “Ste11’s partner”
s.swapcase()
s.split()
s= str(1.2)
```

a method to split a string wherever there is white space

Each type of object has its own special commands called methods.

Use `dir(s)` to see all associated methods and attributes.

Displaying strings

You can add all Python objects including strings:

```
a= 3.14
print('pi= ' + str(a))
print('pi=', a)
```

Strings can be formatted:

```
f"pi is {np.pi:.5f} to five decimal places"
```

where `{}` is a placeholder, where a value will be inserted

`{:1d}` means insert as a one-digit integer
`{:.5f}` means insert with 5 digits after the decimal point
`{:.5e}` means use scientific notation

You can print too

```
print(f"pi is {np.pi:.3e}")
```

For manipulating data use array objects

A type of list for numerical computations from the NumPy module

```
a= np.array([1,2,3])
a= np.arange(10)
a= np.arange(1,10)
a= np.arange(1,10,2)
a= np.linspace(1,100,10)
a= np.logspace(1,3,4)
```

Defining arrays

There are many ways to create an array

```
a= np.ones(4)
a= np.zeros(4)
a= np.empty(4)
```

Arrays can also be multidimensional

```
a= np.ones( (2,4) )
a= np.array( [ [1,2], [3,4] ] )
```

Use

```
a.shape
```

to see the shape of an array in rows and columns

Vectorizing (fast) calculations

`Numpy` applies a mathematical operation to each element of an array.

For example,

```
data= np.linspace(1,100,200)
sindata= np.sin(data)
```

will calculate the sine of each element of the array.

You can also use

```
data*data
2*data
data + data
data**3
2**data
```

Visualising data

`Matplotlib` is the module most often used for plotting:

```
import matplotlib.pyplot as plt
%matplotlib inline
```

An example:

easy to forget

```
x= np.linspace(1, 1.0e4, 100)
plt.figure()
plt.plot(x, np.sin(x), ‘r.-‘, label= ‘sine’)
plt.plot(x, np.cos(x), ‘o’, label= ‘cosine’)
plt.title(‘sine and cosine’)
plt.xlabel(‘x’)
plt.ylabel(‘y’)
plt.legend(loc= ‘upper left’)
plt.show()
```

You can also use `plt.xscale(‘log’)` and `plt.xscale(‘linear’)` to change scales and `plt.xlim([min, max])` to change the limits of the axis.

More on vectoriziation

Another example, calculating a standard deviation:

```
data= np.linspace(1,100,200)
var= np.mean( (data - np.mean(data))**2 )
```

although `np.var` also exists.

Note that `a` and `b` must be the same shape for commands like `a + b` to work otherwise

```
ValueError: operands could not be broadcast together
```

is generated. Use

```
a.shape
b.shape
```

to diagnose the error. You can use `np.reshape` sometimes to fix things.

Accessing elements of arrays

To access a particular element of an array, use square brackets

```
a= np.ones(4)
a[0]
a[2]
a[-1]= 0
```

For a multidimensional array

```
a= np.array( [ [1,2], [3,4] ])
a[0,0]
a[1,2]
a[1,2]= -1
```

Slicing arrays

To access a range of elements of an array, we use slicing

```
a= np.eye(5)
a[0, :]
a[:, 1]
```

The syntax is

```
start index : end index:  stride
```

so

```
a[1:3, :]
a[:-1, 0]

b= np.arange(20)
b[2:12:3]
b[::2]

a[1:4:2, 1]
```

are all valid.

Selecting subarrays

You can use an array to access elements of an array:

```
a= np.arange(20)
theseones= (a < 10)
a[theseones]
```

or in one command

```
a[ a < 10 ]
```

Similarly, you can use

```
a[ a == 4 ]
```

where

```
a == 4
```

tests all elements of `a` to determine if each is equal to `4`

Defining row and column arrays

To define a 1-dimensional row, use

```
a= np.ones((1,4))
```

To force a 1-dimensional array to be a row array, use

```
a= b[None,:]
```

Similarly, to force a 1-dimensional array to be a column array, use

```
a= b[:,None]
```

Other Python objects

List objects

```
c= [1, ‘hello’, 3.0, ‘a’]
c.pop()
c.append(4.5e5)
c= []
```

Tuple objects

Tuples are like lists but cannot be changed

```
c= (1, ‘hello’, 3.0, ‘a’)
```

Loops: repeating tasks

To perform the same or a similar task multiple times, we use loops.

For example,

```
data= ['Ste11  2.3', 'Fus3  0.1', 'Ste12  9.8']
for d in data:     ← note the colon
  print(d)         ← you must indent
```

The variable `d` takes each value in `data` in turn.

A more complicated example extracts the numerical value for each gene:

```
m= np.empty(len(data))    ← predefine the array to store the numerical data
i= 0
for d in data:
  ds= d.split()
  m[i]= float(ds[1])
  i += 1
```

increase `i` to store the data in the next element of the array → `i += 1`

note all the commands in the loop are indented

More loops

We can shorten the code with `enumerate`:

```
m= np.empty(len(data))
for i, d in enumerate(data):
  ds= d.split()
  m[i]= float(ds[1])
```

The variable `i` takes the index for the element in data that is currently in the loop.

Loops can be nested:

```
for y in np.arange(1970, 2002, 2):
  for m in ['Jan', 'Feb', 'Mar']:
    print(m, str(y))
```

Branching with `if` statements

To perform a check on a quantity and then execute different actions depending on the results, we use `if` statements:

```
for d in data:
  if d > 100:
    print(‘high’)
  elif d > 50:
    print(‘medium’)
  elif d > 10:
    print(‘low’)
  else:
    print(‘error’)
```

note the colon (arrow to `if d > 100:`)

you must indent (arrow to `print(‘high’)`)

The logical expressions tested can be more complex:

```
if (a.shape[1] == 100 and a[0] > 0):

if (a > 0 or b > 0):
```

Writing functions

A function is an independent piece of code that can take inputs and produces outputs.

Example 1

To define a function you need `def`, brackets and a colon and use indentation.

```
def printdays():
  for d in ['Mon', 'Tue', 'Wed', 'Thu', 'Fri']:
    print(d)
```

Example 2

Using an input (a function can have any number of inputs)

```
def printerrors(d):
  error= np.std(d)
  print('d=', np.mean(d), '+/-', error)
```

Example 3

With an input and an output

```
def distance(d1, d2):
  dis= np.sum((d1 - d2)**2)
  return dis
```

Note that we've assumed that `d1` and `d2` are `NumPy` arrays. Better code would be

```
def distance(d1, d2):
  d1= np.asarray(d1)
  d2= np.asarray(d2)
  dis= np.sum((d1 - d2)**2)
  return dis
```

Example 4

With optional inputs

```
def scatter(d1, d2, marker= '.'):
  if len(d1) == len(d2):
    plt.figure()
    plt.plot(d1, d2, marker= marker)
    plt.show()
    return np.corrcoef(d1, d2)[0,1]
 else:
    print('Arrays must have the same length')
    return False
```

better to always return a result then return in only one case → (return False)

Calling `scatter(d1, d2)` uses a dot to plot each data point; calling `scatter(d1, d2, '+')` uses a cross as does `scatter (d1, d2, marker= '+')`.

Modules

Modules are a single file with a collection of functions.

To use your own module, there are several options:

```
import mymod
mymod.myfunction()

from mymod import myfunction
myfunction()

import mymod as mm
mm.myfunction()
```

If you edit your module, you need to `reload` it for the changes to take affect

```
import mymod

from importlib import reload
reload(mymod)
```

Navigating directories

In `IPython`, you can see the current directory with

```
pwd
```

You can change into a new directory with

```
cd newdirectory
```

and move up a directory with

```
cd ..
```

To see the contents of directory, use

```
ls
```

To access a directory in your home directory from anywhere, use

```
cd ~/newdirectory
```

De-bugging and errors

`NameError:`	used an undefined variable
`SyntaxError:`	mistyped a Python command
`ImportError:`	Python cannot find a module you wish to import
`AttributeError:`	mistyped the sub-command of a Python object
`IndexError:`	tried to access part of an array or list that doesn't exist
`TypeError:`	called a function with the wrong type of argument

nan: not a number

If you try and perform a mathematical calculation that returns infinity, such as

```
np.log(0)
1/0
```

`NumPy` will return

```
np.nan
```

which stands for "not a number".

If you get `nans` as an answer, check to see if you are dividing by zero or taking either the logarithm or square root of zero.

Magic commands

Magic commands are `IPython` commands and are prefixed by `%`

- `%reset :` `IPython` forgets all variables
- `%run :` run a script
- `%paste :` paste text preserving spacing
- `%pdb on :` switch on Python debugger

Exercises

1. Calculate the value of the normal distribution when `s=2`, `m= 0.1`, and `x= 1`

$$\frac{e^{-\frac{(x-m)^2}{2s^2}}}{\sqrt{2\pi}s}$$

1. Using a `while` loop to make a table where the number of molecules (from 1 to 10) is printed side-by-side with their concentration in bacteria.

2. With the dataset `data` show below

   ```
   data= ['GAL1', 10, 'GAL2', 0.1, 'GAL3', 0.05, 'GAL7', 0.4]
   ```

   write a `for` loop that prints each gene beside its corresponding value.

3. Use a `for` loop to sum `1/i` for all the numbers `i` ranging from 0 to 100.

1. Plot in the same figure *sin(x)* and $sin^2(x)$ for x between 0 and 10. Add a legend and a title to your figure.

2. Plot the fraction of activated protein predicted by the Monod-Wyman-Changeux model (Eq. 49) for *n*=1, 2, 4, and 8. Use `subplot`, and plot *n=1* and *n=2* on one subplot and *n=4* and *n=8* on the other. Add legends to each subplot and label the axes.

1. Write a function to convert numbers of molecules to concentrations in bacteria.

2. Write a function to calculate the mean of a single column of numbers.

1. Solve, Eq. 127,

$$\frac{dy}{dt} = kp + f\frac{y^n}{K^n + y^n} - y$$

for y after 100 time units assuming that *f= 40*, *n=5*, *K= 20*, *k= 0.2*, and *p = 0.1*. Plot y versus time.