PHAS0030: Computational Physics
Session 1: Review: arrays, loops, functions

David Bowler

In this session, we will recap the elements of Python that you should know, practise them, and extend them: NumPy arrays and manipulation; plotting with Matplotlib; loops; functions; and documentation. There is also supplementary information on the idea of testing your code, and writing code with tests from the start, and version control, giving a brief overview of `git`.

Objectives

The objectives of this session are to:

- review and refresh your knowledge of Python from PHAS0007
- understand the creation and manipulation of NumPy arrays
- review the use of Matplotlib
- become confident in the use of loops and other flow control constructs (`for`, `while`, `if`)
- recall how to write functions with docstrings, and how data is passed into and out of functions

Review of python

In PHAS0007 you learnt the basics of Python[1], particularly with reference to the use of NumPy and Matplotlib for analysis and plotting. You also learnt about writing functions, and applied this to a project involving projectiles. The visualisation package used in the last few sessions (`vpython`) will not concern us here, but all of the other ideas will form the basis of this course. You are **strongly** encouraged to make sure that you are comfortable with what you learnt in PHAS0007.

For those who have previously used Python, it is important that you adhere to the coding techniques taught in this course, and do not go beyond them. This means not using any of the object-oriented features of Python, nor any libraries or modules beyond those introduced here.

[1] Here and elsewhere we will mean Python 3 when writing Python; Python 2 is no longer under active development.

Editors vs interactive sessions

You will have encountered at least two approaches to writing Python code: the Spyder IDE (integrated developer environment), which consists of an editor and some facilities to run code; and the Jupyter notebook, which enables you to combine text, Python code and the output from that code. The Jupyter notebook is a framework for various languages; we will be using IPython, which is an enhanced version of Python designed for interactive work.

The Jupyter notebook is an extremely valuable environment for learning to code, as well as exploring new ideas, and writing documents that combine text and code. However, it is not really suitable for large-scale programming, and you should be familiar with the use of an editor to write Python files, and a console to run them and analyse the output. While you are free to choose an editor from the many that are available, you are encouraged to be familiar with both Spyder and Jupyter notebooks as they will be the only interfaces available during the exam.

All of the work in this course will rely on a set of Python libraries developed for scientific computing (SciPy, https://scipy.org, is the overall system), specifically NumPy, numerical python, and Matplotlib, a plotting package.

You can get detailed help on NumPy functions in an interactive session in various ways: running `np.function?` will give the documentation; in Jupyter, shift-tab when the cursor is anywhere in the function name or arguments will give a small help window, while shift-tab-tab will give a large window with the function documentation.

Reminder of previous work

In the previous course, you learned various parts of Python which we will recap and deepen in this session. Here we remind you of several important basic ideas (you should ensure that you are happy with the contents of the course before this course). You will practise these concepts later in this session.

Imports

At the start of an interactive session, or any program, you need to *import* various modules. Typically you will always import NumPy and Matplotlib:

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

Notice that we import and rename the modules; this is the accepted practice within the community, and you should follow it. Among other reasons, it makes clear where functions come from[2]. For an interactive session (IPython in a terminal or a Jupyter notebook) you

[2] You should *never* import a module using syntax like: `from numpy import *`, as it will lead to confusion and bugs.

will also need to use the appropriate IPython magic[3] command: `%matplotlib inline` (though there are other options, which you can list with `%matplotlib -l`).

Documentation

You *must* document your code: even well-written code without comments is very hard for others to read, and frequently hard even for the code author after some time. You have encountered three forms of documentation so far: comments; docstrings; and Jupyter notebook text cells. There are some guidelines on documentation along with coding style in Section 7.

Comments, which start with a hash symbol (#) in Python, should appear in the code itself, and describe what blocks of code or individual complex lines are doing. If you are including physical constants or data, you should give the units in a comment. Commenting is, to some extent, a personal preference, but there are key ideas:

- Do not over-comment, as it makes code hard to read
- Do not under-comment, as code is hard to understand
- Most comments should appear on a separate line *before* the code they refer to
- Short comments can come at the end of a line; consider aligning short comments within an area
- Ensure that comments do not contradict the code (keep comments up to date)

Docstrings appear at the start of the definition of a function, and are by convention enclosed with three sets of quotes (`"""` at the start and end of the docstring. The docstring should summarise the purpose of the function, as well as giving details of expected inputs and what output is provided. **Every** function that you write in this course should have a docstring, no matter how trivial it is.

The final form of documentation that you encountered is the *Jupyter notebook text cell*, which allows simple formatting (via Markdown) and inclusion of mathematics (via LaTeX). The purpose of a Jupyter notebook is to produce a stand-alone document, so the text cells should be used to explain the background physics and the purpose of the programming. Anything directly related to the code should appear in the code (imagine, for instance, that you developed some code in a notebook, and the copied it into a larger program: you would lose the information in the text cells).

Control

The heart of any computer program is a set of control statements that manage the flow of the program. You learnt about two forms of loop: `for` and `while`; and a conditional: `if`. Loops allow you to repeat an operation for a set number of times, or while some condition

[3] This is *not* a Python command; it is specific to IPython, and is a convenient short-cut only.

is true. The conditional `if` allows you to execute different operations depending on the state of the code or the value of a variable.

Arrays

It is very often useful to store a series of objects in a single object (e.g. a set of numbers—points on an axis, etc.). Python natively has two ways to do this: lists and tuples; you only met lists in the previous course. A list typically consists of the same type of object, and is enclosed in square brackets[4].

NumPy defines a different way to do this: the array. We will review NumPy arrays in detail below, but you should recall that they are created with special commands (e.g. `np.zeros`, `np.arange`) and can have multiple dimensions.

Plotting

Matplotlib is a powerful plotting tool which is frequently used in Python. After importing the `pyplot` interface to Matplotlib as above, we can plot simple graphs with the command `plt.plot(x,y)` where `x` and `y` are arrays. The plot can be easily decorated with a legend, axis labels, a title, and different point and line styles. Another command which is very useful is `plt.imshow(array)`, which allows you to represent a 2D array.

Functions

Functions are at the heart of programming, and computational physics, and allow code to be broken up into smaller pieces which can be used multiple times. In Python they are defined with:

```python
def function_name(arg1, arg2, ... ):
    """
    Docstring which can be split
    over multiple lines
    """
```

Once a function has been written, it is called with `function_name(var1, var2, ...)`.

Arrays in NumPy

Most of our work in this course will use NumPy, and specifically we will store data in NumPy arrays. These are different to standard Python lists, and NumPy has been highly optimised to make function calls that operate on arrays very fast. Even though Python is an interpreted language (i.e. one that is run through line-by-line by the Python interpreter, unlike compiled languages that create new executable files), the NumPy array operations are almost as fast as optimised, compiled code.

[4] `a = [5, 6, 7, 8]` is a list; we access elements in the list with an index *starting from zero*: `a[3]` would return the value 8.

There are many important commands that you will need to be comfortable with using in order to be a competent Python-based computational physicist. The exercises that follow this section will take you through them all; for now, here are some highlights:

- Array creation, mainly aiming for specific shapes: `np.zeros`, `np.ones` (remember that shape is specified as `(x,y)` etc: `np.ones((5,5))` would give a 5×5 array of ones; note the double parentheses—the function call requires a set of parentheses, and we pass a tuple as the argument)

  ```
  a = np.zeros(10)
  print(a)
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  ```

  ```
  b = np.ones((2,5))
  print(b)
  [[1. 1. 1. 1. 1.]
   [1. 1. 1. 1. 1.]]
  ```

- Array creation, to create a given number of samples between end-points: `np.linspace(start, stop, number)` (`number` is optional, defaulting to 50)

- Array creation, to create an array with a given spacing: `np.arange(start, stop, step)` (both `start` and `step` are optional, defaulting to zero and one, respectively; note that the array *excludes* the final value)

  ```
  a = np.linspace(0,1,11)
  print(a)
  [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
  ```

  ```
  b = np.arange(0,1,0.1)
  print(b)
  [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
  ```

- Vector operations: `np.dot(a,b)`; if both are 1-D arrays, `np.dot` is equivalent to a scalar product; if they are 2-D arrays, it is equivalent to matrix multiplication. Also: `np.cross`, `np.outer` (the outer product of two vectors gives a matrix: $\underline{a} \otimes \underline{b} = \underline{\underline{A}}$, with $A_{ij} = a_i b_j$) on two arguments `a` and `b`.

- Array slicing and stepping: `a[start:stop:step]` selects elements from start to stop with spacing between them of step (an array can be returned in reverse order using a step of -1: `a[::-1]`)

  ```
  a = np.arange(0,6)
  print(a)
  [0 1 2 3 4 5]
  ```

  ```
  print(a[0::2])
  [0 2 4]
  ```

```
print(a[1::2])
[1 3 5]
```

- Combining arrays: `np.append(array, values)` returns a *new* array with the array-like object `values` added to the end of `array`; `np.insert(array,index,values)` inserts `values` into `array`, at the position before `index`, and returns a *new* array; `np.concatenate((a1, a2, ... ))` will return a *new* array formed of the concatenation (joining end-to-end) of the arguments `a1`, `a2` etc

- Reshaping arrays: `np.reshape(array,shape)` takes `array` and reshapes it according to the tuple `shape`

```
a = np.arange(0,6)
print(a)
[0 1 2 3 4 5]

b = np.reshape(a,(2,3))
print(b)
[[0 1 2]
 [3 4 5]]
```

- Shifting all entries: `np.roll(array,shift,axis)` adds `shift` to the index of each entry in `array` (it shifts to the right or left). For multi-dimensional arrays, `axis` can be specified.

```
a = np.arange(0,6)
print(a)
[0 1 2 3 4 5]

b = np.roll(a,1)
print(b)
[5 0 1 2 3 4]

b = np.roll(a,-1)
print(b)
[1 2 3 4 5 0]
```

2D Arrays

The power of NumPy comes from its ability to work on arrays: it is much more efficient to operate on an entire array than to do it element by elements[5]. If we are modelling a two dimensional problem, then this presents a challenge. Let's say that we want to calculate the electrostatic potential due to a charge at every point in a square. The final result will be a 2D array, and will require arrays of x points and y points.

[5] For relatively small problems, you won't notice this; however, for large problems it makes a huge difference

If we were to use `for` loops (explained more below) and 1D arrays for x and y, we would write:

```
for xpt in x:
    for ypt in y:
        potential[i,j] = 1.0/np.sqrt(xpt*xpt + ypt*ypt)
```

where we have not discussed how the indices for potential (`i` and `j`) are defined. But this implementation using x and y as 1D arrays does not allow us to use NumPy's efficiency: instead, we need to be able to write an expression that is something like:

```
potential = 1.0/np.sqrt(x*x + y*y)
```

where `potential`, `x` and `y` are all the same kind of variable (in this case a 2D array). Notice how much more compact and clear this is in terms of code.

So really we need the array `x` to be a 2D array, defined over the whole grid, but giving only the x-coordinate for each point; similarly for the array `y`. This involves some redundancy (the arrays will have columns or rows that are the same) but the efficiency gain is well worth it.

Numpy has a routine that will generate appropriate 2D arrays from two 1D arrays: `np.meshgrid`:

```
x2D, y2D = np.meshgrid(x, y)
```

will return two 2D arrays (`x2D` and `y2D`) which can be used for efficient NumPy calculations.

Exercises

In class

You should be able to finish these exercises in the class; if you do not, make sure that you finish them in your own time.

1. There is an example above that creates arrays from 0 to 1 with a spacing of 0.1 using both `np.linspace` and `np.arange`. Write some Python code to create arrays between two points, a and b, with a given step size, using both `np.linspace` and `np.arange` (you will have to work out an appropriate number of points for `np.linspace`).

2. Now write Python code to create arrays between two points, a and b, with a given number of points, using both `np.linspace` and `np.arange` (you will have to work out an appropriate step size for `np.arange`). Be sure that you understand the differences between these functions and these approaches.

3. Create an array of integers from 0 to 49 inclusive. Use array slices to print the first five and the last five entries. Use array stepping to print every five entries (experiment with the starting point as well).

4. Operating on the array from the previous question, use `np.roll` to shift the array to the left and right by five places. Find a way to produce the same effects with array slices and `np.append`

5. Use the array stepping at the end of question 3 and use `np.reshape` to create a (2×5) array containing only multiples of five

6. Use `np.reshape` to make a (10×10) array from an array from 0 to 99 inclusive. Now use array slicing to print rows and columns (be sure to understand the difference !). Also use array slicing to print out some (2×2) sub-matrices (experiment).

Further work

These exercises are an important part of this course, and will deepen your understanding of Python and NumPy. You should make sure that you attempt all these exercises, otherwise you will not understand the material fully. Ensure that you have finished the in-class work first.

1. Explore how the normal arithmetic operators + and * work with arrays, and compare to `np.dot`. You should choose small arrays with integer spacing so that you can understand what is being done.

2. Create an (8×8) array of zeros, and then change alternating entries to 1 using slices and steps. Plot with `plt.imshow`: you should see columns of colour.

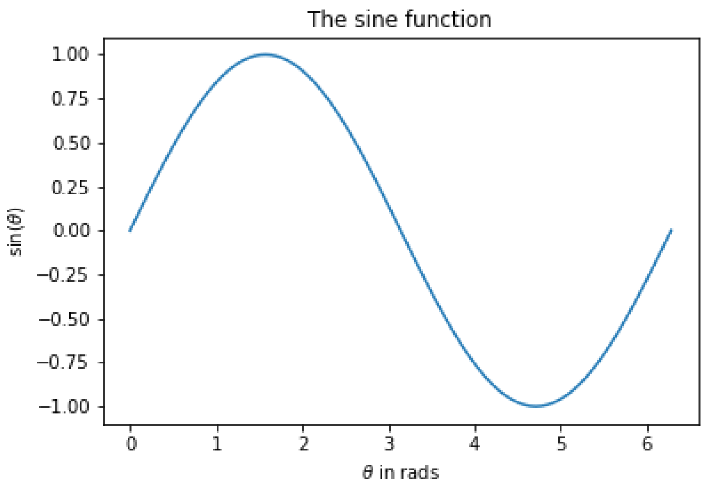
3. Still setting different elements to 1, now try to create a chessboard from the array in the previous question by treating even and odd rows differently.

Plotting with Matplotlib

Effective representation of scientific data is an important part of all physics, but particularly computational physics. Matplotlib is an excellent package that allows you to produce graphs, contour plots and three dimensional images. You should always remember that content is more important than presentation, and that it is very easy to spend more time than is wise creating images that are perfect (but whose perfection adds little to their value!).

The simplest approach to using Matplotlib is what you learned in the previous course: using the `pyplot` module, which is imported as `plt`. Changes can be made to the current figure, but there is little control. An example can be seen in Fig. [fig:SimplePyplot].

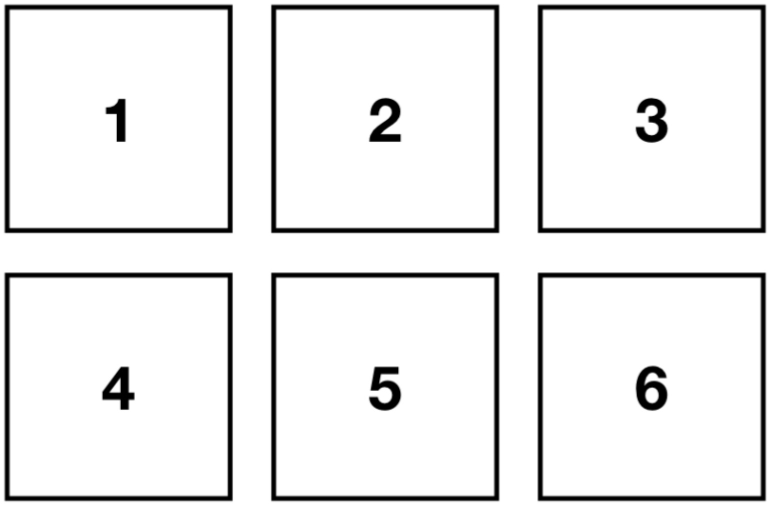
```
x = np.linspace(0,2*np.pi,101)
plt.plot(x,np.sin(x))
plt.xlabel(r"$\theta$ in rads")
plt.ylabel(r"sin($\theta$)")
plt.title("The sine function")
```



The simple `pyplot` approach and output.[fig:SimplePyplot]

There is a more powerful approach where individual figures are created and can then be controlled separately (so that, for instance, we can create arrays of figures). In this case, we start by creating a figure (something that can hold one or more plots) and then populate it with subplots:

- Create the figure: `fig1 = plt.figure()` The figure can now be referenced, and can be displayed again just by giving its name
- Sub-plots (or, if you like, graphs) are called *axes* in this approach. They are added as follows; note that we keep a reference to the sub-plot: `ax1 = fig1.add_subplot(rcn)` This adds a graph as part of a 2D matrix of subplots with `r` rows and `c` columns at position `n` (where positions start at 1 and increase row by row)



An array of sub-plots with two rows and three columns. This would be created with `fig.add_subplot(23n) with n=1\rightarrow 6`

[fig:S1_subplot]

- Plot data on one of the axes: `ax1.plot(x,y)` This is the same function as use in the simple `plt.plot` call

- Annotate each axis as desired, using calls such as `ax1.set_xlabel()`, `ax1.set_title()`

You should ensure that your graphs always have labels for each axis as well as a title (which should not just repeat the axis labels!) and that lines are correctly labelled with a legend, if appropriate. We will explore more powerful and complex use of Matplotlib throughout the course.

We can plot two dimensional data using colours (`plt.imshow` or `ax1.imshow`) though you should be careful with the choice of colormap: rainbow-type colormaps are well known to be a poor choice[6] and you should *not* use them; Matplotlib now defaults to viridis, though bone and YlGnBu are also effective.

[Reminder: within `plot` you can specify the plotting style with a compact notation, e.g. `'bo-'`. Colours (red, green, blue, cyan, magenta, yellow, black, white) are given by their first letter except for black (`k`); you can also set linestyles (`-`, `–`, `:`, `-.`) and markers (`o`,`+`,`*`, `h`, etc). You set labels with `plt.xlabel` etc.]

Exercises

In class

1. Practise plotting simple functions (trigonometric etc) using the basic pyplot approach, adding labels to axes, different lines with colours and symbols.

2. Now experiment with creating a figure: in the first place, try two graphs showing sine and cosine. You may want to give a figure size when creating the figure (`plt.figure(figsize=(w,h))` where width and height are given in inches).

3. We are going to develop a 2D array of distance, following on from the exercise in 3.2. Create 1D arrays of x and y from 0 to 9 with a spacing of 1, and using `np.meshgrid`, create equivalent 2D arrays. Use these to calculate the distance of all points in the grid from the origin, and plot it using `plt.imshow`. Experiment with different colour maps, and try creating more arrays with different grid spacing.

Further work

It is often said that sunflower seeds in a flower follow a Fibonacci sequence[7]; in this exercise, you will explore what this means.

[6] This has a certain amount to do with perception both in terms of ordering and spacing, but also with mapping to greyscale; you can find more in various places, for instance https://bids.github.io/colormap/, or read the Matplotlib tutorial.

[7] The rule for this sequence is $f_m = f_{m-1} + f_{m-2}$; this gives a sequence 1, 1, 2, 3, 5, 8, 13, …

The key parameters that you will need to explore in polar coordinates are:

$$\begin{aligned} r &= 0.5\sqrt{n} \\ \theta &= \frac{2\pi n}{1+\phi} \\ \phi &= 0.5\left(1+\sqrt{5}\right) \end{aligned}$$

where n is an integer giving the number of the sunflower seed.

1. Set up an array for n (integer steps from 0 to anywhere between 100 and 1,000 will be fine, though you may experiment) and use this to create arrays of r and θ for the seeds.

2. Now calculate arrays of the cartesian coordinates for the seeds using the standard conversion from 2D polar coordinates.

3. Make a plot showing the locations of the seeds using just points (I found that using `plt.plot(x,y,'o')` worked well). Can you see any patterns?

4. You will now need to change the plot to explore how Fibonacci numbers appear; you will find that plotting every f seeds makes the patterns much easier to see. Start with $f = 2$, and use array stepping to plot every 2 seeds (e.g. `x[::2]`). If you use two plot commands (on the same plot) for every 2 seeds starting at 0 and at 1, you should see a pattern emerging.

5. Do the same thing, but now use `plt.polar(theta,r,'o')` (or whatever style you want). Note that the order of variables in the call to the polar plot is *not* the scientific standard !

6. Now try $f = 5$ as the Fibonacci number: you will need to have five plot commands, starting with a different number in the arrays each time. Try it with a `for` loop. [Hint: use `x[start::step]` to do this]

7. Try to create an array of subplots for the first few Fibonacci numbers (avoid 1; I found that the first six numbers all give interesting plots). You will need two `for` loops: one for the Fibonacci numbers, and one for the different coloured points

Loops and control

Repeating a set of operations for different input values is really at the heart of most computing. In this section we will discuss the two main loops available in Python, as well as other ways to manage the flow of the program.

For loops

If you have a set of items, either in a sequence or in some container such as an array, and you want to iterate over them, then you should use a `for` loop. If you have an array `a`, then you can iterate over the members of the array with `for member in a`, in which case the loop will repeat with the variable `member` set to each of the individual members of `a`. Remember that only lines that are indented relative to the `for` statement are controlled by the loop.

```
for i in range(5):
   print(i)
0
1
2
3
4
```

You can easily iterate over a range of numbers, using the `range` function, for instance as in Fig. [fig:range]. Of course you can tailor the sequence: `range(start, stop, step)`, where `start` and `step` are optional. An excellent example of a mathematical operation that maps onto a `for` loop is a sum, for instance the exponential:

$$e^x \simeq \sum_{i=0}^{N} \frac{x^i}{i!}$$

could be written in Python as:

```
from math import factorial
# Assume x and N have been specified elsewhere
sum = 0.0
for i in range(N+1):
    sum += x**i/factorial(i)
```

If you want to iterate over an array while also keeping track of the index, you could define and increment an index:

```
a = [5, 6, 7, 8]
i = 0
for v in a:
    print(i,v)
    i += 1
```

gives:

```
0 5
1 6
2 7
3 8
```

There is nothing wrong with this approach, but it risks errors if the index is not zeroed or updated. It is much simpler to use the command `enumerate`, which returns both the *index* and the *value* for each entry in an array[8].

```
a = [5, 6, 7, 8]
for i, v in enumerate(a):
    print(i, v)
```

also gives:

```
0 5
1 6
2 7
3 8
```

Loops can also be nested, with indentation controlling which loop controls which statements, as shown below; note that I have not written code, but used comments to indicate control.

```
for i in range(5):
    for j in range(5):
        # Lines repeated for both j and i
    # Lines repeated just for i
# Lines outside the control of either i or j
```

There is a simple, elegant way to create lists and arrays which require some kind of function evaluation, called a list comprehension. A simple example is:

```
[x*x for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The basic syntax gives the operation to be carried out (in this case `x*x`) followed by some loop operations. Note that this is identical to the explicit form:

```
a = []
for x in range(10):
    a.append(x*x)
print(a)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The list comprehension is much more concise and readable, and we can enclose the result in a numpy array quite simply: `a = np.array([x*x for x in range(10)])`. We can also add conditions in the comprehension (see Further Work, below).

[8] Note that the use of `enumerate` with arrays with more than one dimension needs a little care, as it will return sub-arrays.

Exercises

In class

1. Loop over two variables from 1 to 5 and store the product of the numbers in a 1D array or list (you may want to create a NumPy array in advance, or use a Python list and append)

2. Now write a list comprehension to do the same thing

3. Use `np.reshape` and the list comprehension to store the result in a (5×5) array

4. Return to your `for` loops; add a condition so that you only store the product if you are in the lower triangle of the matrix (you can write: `a = x*y if condition else 0` to do this - and you can do this inside the argument to a function…)

5. Following on from the last in-class exercise in Section 4.1, we are going to use `for` loops to create the distance array, with an addition. Create a 2D array filled with zeros to store the distances. Using explicit loop indices (say `i` and `j`) and the same 1D x and y arrays as before, use two `for` loops to calculate the distance from each point in the grid to an *arbitrary* point $\mathbf{r}_0$, defined in terms of the variables `x0` and `y0`. For a grid spacing of 1, set $x0 = 4.5$, $y0 = 4.5$ and create and plot the distance array.

6. Now use enumerate to do the same job without loop indices.

While loops

A `while` loop allows you to repeat an operation while a given condition is true, for instance:

```
i = 0
while i<5:
    print(i)
    i += 1
```

will give the exact same behaviour as seen in Fig. [fig:range], though of course much more complex behaviour is possible. We often use a while loop where convergence to a certain limit is required (e.g. in an expansion of a function, we add terms while they are larger than a threshold).

It is important to ensure that a `while` loop does not go on forever: some form of counter is necessary to make sure that the loop stops after a set number of iterations, even if the condition is not fulfilled. This can be done simply, for instance: 3cm

```
i = 0
a = 1.0
while a>0.0 and i<100:
    i += 1
    a /= 2
```

Try this in an IPython session, and explore what happens if you increase the allowed number of iterations (print a at each step to see the behaviour).

More control statements

There are important extra statements that you have not met yet that allow you to control loops further:

- `break` exits the innermost loop of any set (this means that any remaining iterations will *not* be carried out)
- `continue` moves to the *next* iteration of the present loop
- `else` allow you to add statements to run if the loop is exhausted (`for` loops) or for a false condition (`while` loops). Note that it is *not* executed following a `break` statement.

Conditions and branching: if

We can set up branches which depend on conditions using `if`, `elif` and `else`. You have met these in the previous course. There can be zero or more `elif` clauses, and the `else` clause is optional. You should try to avoid making large numbers of different possible cases, as this can be hard to read.

Conditions (which are used both for `if` and `while`) can be combined (using `and` and `or`) as well as negated (`not`). Comparisons can also be combined (e.g. `a < b < c`). Use of parentheses (round brackets) is advised for clarity when making complex conditions.

```
if (0.0 < x < 1.0) and (i < 100):
    # Do something
elif (x < 2.0) or (i==100):
    # Do something else
else:
    # Do final thing
```

Exercises

In class

1. Write a `for` loop to calculate the squares of integers from 1 to 20, but only if they are even (include an `if` and an `else` clause and use `continue`; you may find the modulus operator % to be useful).

2. Write a short piece of code to test for prime numbers for integers from 1 to at least 100. Only write out a number if it is prime. (You will need to use `for`, `while` and `break`; remember that you only need test a number n for factors up to $\sqrt{n}$.)

3. Create a (2×5) array containing the numbers from 0 to 9 inclusive. Iterate over the array using `enumerate` and print what is returned. Now introduce another loop also using `enumerate` to iterate over what is returned from the first `enumerate`. Print both indices: what is happening ?

Further work

Creating a Mandelbrot set image is relatively simple, and gives good practice with flow control and arrays. The further work will concentrate on this.

The Mandelbrot algorithm is applied to points in the complex plane $c = x + iy$ and applies the following iteration: $z_{n+1} = z_n^2 + c$ with $z_0 = 0$. Points which have $|z_n| \leq 2$ as $n \to \infty$ do not diverge, and are within the set; we will replace ∞ in the divergence criterion with $n \geq 70$. (Note that, while complex arithmetic is possible in Python, we will use real arithmetic here.)

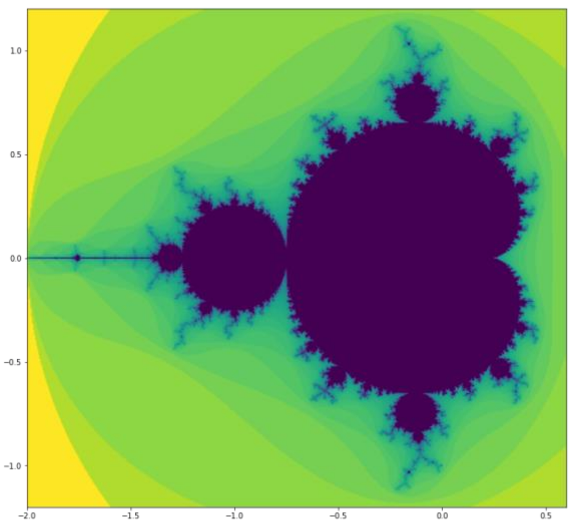
1. You will need to create two 1D arrays for the real and imaginary parts of the complex number c, with ranges $-2 \leq x \leq 0.6$ and $-1.2 \leq y \leq 1.2$

2. You will then need to create a 2D array of zeros which has the shape $(nx \times ny)$ to store the results

3. You should write two `for` loops over x and y using `enumerate` so that you can store the result of each iteration in the results array, with the appropriate index

4. For each point, use a `while` loop to perform the iteration (use two conditions: one on the magnitude of z, the other on the number of iterations). You will need to evaluate the real and imaginary parts of z_{n+1} separately. You may want to create an array `[x,y]` for z. You should keep track of the number of iterations around the while loop

5. You can make a simple black-and-white image by labelling a point within the set as 0 and outside the set as 1; if you instead store $1 - \sqrt{n/70}$, where n is the number of iterations performed, then the resulting plot using `plt.imshow` is quite attractive, as seen in Fig. [fig:mandel].



An example of the Mandelbrot set

Functions

A function allows us to put a set of statements and operations into a single place, and re-use them on different inputs. This gives much more efficient and readable code, as well as reducing the likelihood of errors (which can easily occur if copying or repeating code).

A function takes zero or more arguments (or parameters), operates on them, and returns zero or more results via the `return` statement. Note that the parameters passed in are *not* changed by the function, and that variables inside the function take precedence over those outside (so if you have two variables with the same name, one inside the function and the other outside, the local variable will be used).

```python
def factorial(n):
    '''Calculates the value of n!  Returns zero for
       negative input and rounds down to nearest integer

       Input:   n (number whose factorial is required)
       Outputs: n!
    '''

    # Catch negative exception
    if n<0:
        return 0
    elif n==0:
        return 1

    # Round down and do first step
    round_n = round(n)
    fac = round_n
    # Loop over remaining terms
    while round_n>1: # So round_n - 1 > 0
        round_n -= 1
        fac *= round_n
    return fac
```

We can pass functions as arguments to another function—this is very useful if writing something like a differential equation solver that should be able to operate on different functions. Several different outputs can be combined in one return statement.

A function should always start with a docstring: the docstring should give details of the expected arguments (and whether they are required or optional) as well as what the function does, and what it returns.

Documentation

The docstring appears at the start of functions (just after the definition line) and is enclosed in triple quotes. All your functions should have a docstring, no matter how trivial: first, because it will get you into the habit of writing them; second, because simple functions can often be developed into more complex functions, which will require a docstring. You can also add a docstring at the beginning of a file (normally this is used to give help on a module—we will discuss modules in Python later). The docstring should give any user of the function *all* the information that they need to use the function: the interactive help in IPython is taken from the function docstring.

Comments should be written *as you write the code*, not as an after-thought. Good commenting should make clear what the code is doing, what unexpected statements are doing (e.g. the `while` condition in the factorial example above) and what physics is being implemented (including units).

A good choice of variable and function names can go a long way to making code readable (also see Section 7 for some information on Python coding style). There is often a temptation to choose simple, short variable names (e.g. `nx` or `i2`) to make writing code quicker; this is almost always a false economy, and descriptive, clear variable and function names are important.

Exercises

In class

1. Write your own version of the factorial function in the notes (do something differently: use `for` in place of `while`, or go up rather than down)

2. Write a function to calculate nth term in the Fibonacci sequence ($f_{n+1} = f_n + f_{n-1}$ with $f_0 = f_1 = 1$. You can do this in several different ways; it is possible to call a function from within itself (this is known as a *recursive* function call) but this can be inefficient. You could start with the first two terms and build up.

3. Continuing the work on distance calculation from the previous sections, you should now write a function to calculate the electrostatic potential due to a charge `q0` at a point (`x0`, `y0`). The arguments should be the x and y coordinates of the 2D grid (from `np.meshgrid`), the position of the charge and the value of the charge. The function should return the potential, and should use the most efficient NumPy approach. You should work in atomic units (so $\phi = q/r$ with an electron having $q = -1$). You might like to think about what happens if the charge is placed directly on a grid point (both from a physics point of view and a coding one!)

4. Plot the potential using `plt.imshow` (add a colour bar using `plt.colorbar`). Check that your result is reasonable.

5. Now calculate the electric field on the grid points due to the charge:

$$\mathbf{E}(\mathbf{r}) = q\frac{\mathbf{r}-\mathbf{r}_0}{|\mathbf{r}-\mathbf{r}_0|^3}$$

6. You should return the x and y components of the field as separate 2D arrays. You might like to add this calculation to your potential function, returning both the field and the potential, or write a new function.

7. Now plot the field using `plt.quiver(x,y,Ex,Ey)` where the arrays are all 2D. (I found that I needed to make the figure square; if you are using `plt` then you can set this with `plt.rcParams['figure.figsize'] = 6, 6`.)

8. An alternative is to use streamlines: `plt.stream(x,y,Ex,Ey)`. You can add an optional argument `density` (where a value of 1.0 gives no change) to increase or decrease the streamline density.

Further work

1. Write a simple function to split an array into even and odd entries, and return two arrays (use appropriate start and step values)

2. Write a simple function to interleave two arrays, returning one array (it should be the inverse of your split function)

3. Write a function to implement the Mandelbrot iteration (it should take two numbers as input, and return the number of iterations required). Check that it works by replacing the heart of your Mandelbrot code from earlier.

Coding style

Coding style is something that can be personal but is also extremely important to make your code readable, both for you, and for others. Almost all code is shared nowadays, and there have been some notable issues within physics where people have refused to share their code, only to find that there were errors leading to anomalies (see, for instance, https://dx.doi.org/10.1063/PT.6.1.20180822a). Writing your code in accepted style will help others to read it.

Python has a defined style, which you can read about: https://www.python.org/dev/peps/pep-0008/. There is also a style guide for docstrings: https://www.python.org/dev/peps/pep-0257/. The key points can be summarised as follows:

```
foo = function(one, two,
               three, four)
```

- Four spaces per indentation level (not tabs)

- Lines should not be more than 79 characters

- Split long lines in parentheses, as shown in Fig. [fig:indent]

- It is possible to use a continuation marker (\), but best not to do this

- Put all imports at the beginning of a file

- Do **not** import all (`from numpy import *`)

- Whitespace can make code clearer (only *after* comma, before and after an operator (+, -, etc.), not before/after bracket)

Comments: examples

Over-commenting: the example below is not much of an exaggeration. By commenting lines whose purpose is obvious, the code becomes harder to read. Note that all comments are in-line, and long enough to overflow. The comments are also not aligned (alignment can help with readability).

```
i = 0 # Set i to zero
tol = 0.1 # Tolerance for while loop I will show next
result = 1.0 # Need to set the value for result large enough that it will not
break the while loop
while result<tol: # While loop will continue while the result is bigger than
the tolerance
    result = function_call() # Call function to update result
    i=i+1 # Add one to counter
```

A better level of commenting for the same source code might be:

```
# Initialise
i = 0
tol = 0.1
# Set result to arbitrary value > tol to pass while condition
result = 1.0
while result<tol:
    result = function_call()
    i=i+1
```

Under-commenting: if variable names are unclear and no context is given, a lack of comments can make code hard to read.

Assignment

The final assignment this week is to create a Jupyter notebook which explores the electrostatic potentials and electric fields of different arrangements of charges: start with a monopole (one charge); add a second with opposite charge (dipole); and end with four (quadrupole, two of each charge arranged in a square). Your notebook should give a brief introduction (ideally with equations for the physics) and include functions and appropriately commented code. You should end with a brief conclusion discussing both the results and what you have learned about representing scalar (potential) and vector (field) quantities.

Progress Review

Once you have finished *all the material* associated with this session (both in-class and extra material), you should be able to:

- Create arrays with several dimensions in NumPy, and initialise them
- Use flow control to repeat and iterate, as well as testing different conditions
- Write Python functions, taking inputs and returning outputs
- Produce simple plots using `matplotlib`

You should also have started to understand the following more advanced concepts (which will not be tested directly, though are important to understand and to do; note that your coding style will be part of assessments).

- Understand how to stage and commit changes to a `git` repository, and then push to GitHub (or equivalent)
- Write simple automated tests for your functions
- Write good-quality code complying with Python style

Python quirks

Python has some features that may confuse new users or users familiar with other languages:

- Zero-based lists and arrays: the first element of a list or an array is `0`, not `1`
- Assignment: `b = a` points a new label, `b`, at the area of memory already labelled with `a`. See an illustration in Fig. [fig:assign].

```
a = [1, 2, 3, 4, 5]
b = a
print(b)
[1, 2, 3, 4, 5]
a[2] = 7
print(b)
[1, 2, 7, 4, 5]
b[3] = 9
print(a)
[1, 2, 7, 9, 5]
```

- String "addition": `s = 'Hello ' + 'world'` gives `'Hello world'` (strings have lots of useful features: you could try `s.center(width)`, `s.lower()`, `s.rjust(width)`, `s.splitlines()`)

- Multiple assignment: `x = y = z = 10`
- Chaining comparisons: `if 3 < x < 4:`
- Whitespace at the start of a line *matters* and indicates control of loops and conditionals (more below, but *be careful*)

Version control

Version control is an important part of modern computing practice, and involves saving (committing) changes to your code regularly and systematically. If you have ever copied files multiple times, renaming each time, to keep track of changes, or ended up unsure of what you changed in a program to make it work (or stop it working) then you may appreciate the need for version control. It will keep track of the *changes* between versions of a file, and so relies on the coder to choose what changes to register. It is *very important* that you only ever commit working code, and that you make useful commit messages to explain what you have done.

At the simplest level, you could do this all on one machine (your personal laptop, say) in which case it would be hard to share your code (or allow others to work on it simultaneously). You would also only be able to work on the code when you had access to that machine. Modern version control systems allow you to share your changes with others using a variety of models; we will use an approach where changes are shared with a central server to which you will send changes. In this model, you work on a local copy of the code, commit your changes, and then push these to the server.

The Git tool

We will be using the version control tool `git`, which was developed to manage the Linux kernel source code, and is now probably the most widely used version control tool. There are many introductions to Git; there are a number of videos on the official page (https://git-scm.com/doc) and a book on Git which is comprehensive (https://git-scm.com/book/en/v2), and free to read and download. However, these notes will contain all the basic commands that you need to use in the course.

GitHub: a development platform

The central server that we will be using is GitHub. You will need to create an account on GitHub, following these steps:

- Register with GitHub: https://github.com/
- Be sure to use your official UCL email address (this enables you to get an educational account)
- Sign up for an educational account: https://education.github.com/ and choose "Join GitHub education"

Concepts in version control

There are various important ideas that you will need to understand in version control. The terms used here are those from git, but the concepts are transferrable.

The repository

This is a storage area for a particular project, where you will put all of the files that you need. You might decide that you want to create a separate repository for each session in this course, alongside a repository for your mini-project, or you could have just two (one for the sessions and one for the mini-project) or any other layout that fits.

Stage or add

When you stage a file or set of files, you are registering the changes between them and the present version as ready to be committed (saved to the repository).

Commit

Creating a new version of the repository that will include all the changes that you have staged.

Push

Make the changes available: push them to the server (the origin).

Once you have a repository set up (details below), you need to develop code and commit. The basic workflow is as follows:

- Make changes to or create files
- Test the code, fix bugs
- Stage changes to repository (also known as add changes)
- When you have a new version, you commit the changes that have been staged
- You push the changes to the central server

Creating and cloning a repository

You can create a repository on a local machine if you choose, but the simplest way to create a repository is probably via GitHub: if you log in there, then the page you first see allows you to create a New Repository. You should try this (you can delete or just ignore the result in the future).

You will need to setup git on your local machine; you can do this with single commands or by editting a file. On DesktopUCL it is easiest to do the latter. You should edit the file `.gitconfig` **on the N: drive** (you can use Notepad or any other editor) and ensure that you have the following lines:

```
[user]
name = Your Name
email = your.name.year@ucl.ac.uk
[core]
editor = notepad
```

if you want to use the Windows Notepad editor to write commit messages. If you leave the editor unchanged it will use the vi editor which can be challenging to learn.

You can then clone the repository (make a copy on your local machine) using the command `git clone RepositoryName DirectoryName`. Note that `RepositoryName` will be something like `https://github.com/YourName/Repository` and you can choose any local directory name that you want. To do this on DesktopUCL you should use the Git CMD or Git Bash applications. Make sure that you change to the N: drive first (in CMD, type `n:`, while in Bash you need `cd n:`). You can also use the Git GUI though I found this unhelpful. We cannot help you with your own machines, but Anaconda does allow git installation and you can download a GitHub GUI if you want. (If you are an Emacs user, then Magit is a very powerful interface.)

Staging, Committing and Pushing

Once you have cloned your repository (which will probably be empty), you should create a file using Spyder (or whatever editor you wish) and save it to the directory. Using Git CMD or Bash you can now try the following commands: `git status` (which will show you that you have untracked files present); `git add FileName` (which will add or stage the file for a commit - saying that you want this file to be included in the next commit); `git commit` (which will commit the changes that you have staged, and will ask for a commit message—this should be meaningful); `git push` (which will push the changes to the server, which is GitHub in this case). Try running `git status` after each other command (e.g. after `git add` and also after `git commit`).

Testing your code

Writing tests for code as you write the code (or even before you write the code!) will help to find bugs and provide ways of ensuring that changes you make do not introduce unexpected behaviour. While writing tests may seem time-consuming and unnecessary, it is a key part of modern software development, and is immensely valuable. Python provides testing as standard modules, and we will use the `pytest` module.

At its simplest, we can write a test function by giving it a name starting `test_`. Within that function, you should have a line (or lines) that state expected behaviour, using the `assert` statement. As a concrete example, we could create a test function for a Fibonacci function:

```
def test_fib():
    for n in range(2,10):
        assert fib(n) = fib(n-1) + fib(n-2)
```

If you have a python file `fibonacci.py` (say) which includes the function `fib` and the test function `test_fib` that we have just written above, then running `pytest fibonacci.py` on the command line will test your function as you have specified (i.e. for all numbers between 2 and 9 inclusive).

More generally, if you simply run `pytest` or `python -m pytest` in a directory, any file with a name starting `test_` or ending `_test` will be searched for appropriate test functions. On DesktopUCL you can do this in the Anaconda3 Prompt (which starts in the N: directory). You can also use `run -m pytest` within an IPython session (e.g. in Spyder). Writing this kind of simple test as you create functions is extremely valuable, and is something that you should get into the habit of doing.

Start with simple tests: create trivial functions (say adding one to a number) and create a test for this. Then deliberately break your function (add two, say) and see what happens with `pytest`. Now try writing tests for a factorial function. Think carefully about what behaviour you would assert.