PHAS0030: Further Practical Mathematics & Computing
Session 1: Review: arrays, loops, functions

David Bowler

January 4, 2019

In this session, we will recap the elements of Python that you should know, practise them, and extend them: NumPy arrays and manipulation; loops; functions; documentation; and plotting with Matplotlib. We will introduce the idea of testing your code, and writing code with tests from the start, and consider version control, learning the very basic parts of git.

Contents

1 *Objectives*

The objectives of this session are to:

- review and refresh your knowledge of Python from PHAS1240 (now PHAS0007)
- understand the creation and manipulation of NumPy arrays
- 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 the use of Matplotlib
- learn the basics of version control (specifically `git`)
- begin to explore ideas of automated testing of code

2 *Review of python*

In PHAS1240 (now renumbered as 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.

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

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.

2.1 *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.

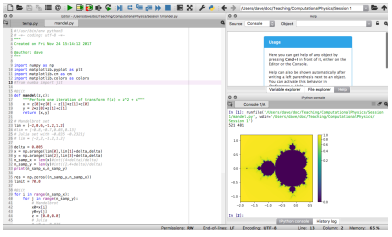

Figure 1: The spyder IDE, showing the editor (left hand column), IPython terminal (bottom right) and help (top right).

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 much of the work in this course will require you to use 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.

2.2 *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 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`).

[2] You should *never* use `from numpy import *`, as it will lead to confusion and bugs.

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

Comments 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.

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

Docstrings appear at the start of the definition of a function, and are by convention enclosed with three sets of quotes (`"""` or

'''). 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 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.

[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.

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. In Python they are defined with:

```
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, ...)`.

2.3 *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. 2.
- 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*)

```
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]
```

Figure 2: An example of how Python assignments work. Notice how both `a` and `b` refer to the same stored data.

3 *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.

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 zeroes; note the double parentheses—the function call requires a set of parentheses, and we pass a tuple as the argument)
- 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)
- Vector operations: `np.dot`, `np.cross`, `np.outer` (the outer product of two vectors gives a matrix: $\underline{a} \otimes \underline{b} = \underline{\underline{A}}$, with $A_{ij} = a_i b_j$)

- 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]`)
- Adding one array onto another: `np.append(array, values)` returns a *new* array with the array-like object `values` added to the end of `array`
- Inserting one array into another: `np.insert(array,index,values)` inserts the new values at the point before `index` and returns a *new* array
- Joining arrays: `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`
- 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.

3.1 *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. Create arrays from 0 to 1 using `np.linspace` and `np.arange`. Experiment with step size and end-point (`np.arange`) and number of points (`np.linspace`) to ensure that you can generate arrays with spacing of 0.1 and 0.01 and with 11 and 101 elements using *both* methods. Be sure that you understand the difference between the two methods.

   [You might like to try to generalise this: define variables to specify lower and upper limits and a step size, and create a general rule for each approach. If you don't have time in class, do this in further work.]

2. Create an array of integers from 0 to 49 inclusive. Print the first five and the last five entries. Print entries with a step of 5.
3. Use `np.roll` to shift the array to the left and right by a certain amount. Find a way to produce the same effect with array slices and `np.append`
4. Use `np.reshape` on the array from 0 to 49 (with slicing) to create a (2×5) array containing only multiples of five
5. Create a 2D array of size (10×10) filled with 0
6. Create an array of integers from 0 to 9 inclusive, and print the outer product of this with itself.

7. Make a (10×10) array from an array from 0 to 99 inclusive, and print rows and columns (be sure to understand the difference !). Print a (2×2) sub-matrix.

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 do 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. Using `plt.imshow` (discussed below in Section 6), plot the array from step 6 above.
3. This is rather blocky; use `np.linspace` to generate an array from 0 to 9 with more points, and plot the outer product of this array with itself
4. 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. Now try to create a chessboard by treating even and odd rows differently.
5. Show that the operation `np.outer` on two 1D arrays of length `n`, `v` and `w`, is equivalent to the simple multiplication of the reshaped array `np.reshape(v,(n,1))` with `w`. Check what `v*w` gives, and that you understand the difference. Choose the arrays carefully.

4 *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.

4.1 *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.

You can easily iterate over a range of numbers, using the `range` function, for instance as in Fig. 3. Of course you can tailor the sequence: `range(start, stop, step)`, where `start` and `step` are

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

Figure 3: Using range to produce a **`for`** loop iterating over the numbers 0, 1, 2, 3, 4.

optional. If you want to iterate over a sequence while also keeping track of the index, you could define and increment an index, but it is much simpler to use `enumerate`, which returns both the index and the value for each entry in an array (Fig. 4). (The use of `enumerate` with arrays with more than one dimension needs a little care, as it will return sub-arrays.)

```
a = [5, 6, 7, 8]
for i, v in enumerate(a):
    print(i, v)
0 5
1 6
2 7
3 8
```

Figure 4: The use of enumerate: iterate over a list and return the index (i) and entry (v).

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)
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).

4.2 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. 3, though of course much more complex behaviour is possible.

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:

```
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).

4.3 *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.

4.4 *Conditions and branching*

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.

4.5 *Exercises*

In class

1. Create a 1D array of integers from 0 to 9 inclusive. Create a (10×10) 2D array of zeroes. Write two for loops to perform an outer product on the 1D array, and store it in your 2D array. Check that it gives the same result at `np.outer`
2. 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)
3. Now write a list comprehension to do the same thing
4. Use `np.reshape` and the list comprehension to store the result in a (5×5) array

5. 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...)
6. Write a while loop to calculate factorial of a number. You will need to think a little carefully about how to do this.
7. Write a for loop to calculate squares of numbers in a certain range, but only if they are even (include an `if` and an `else` clause and use `continue`; you may find % as modulus useful).
8. Write a short piece of code to test for prime numbers for a range of at least 100. Only write out a number if it is prime. (You will need to use `for`, `while` and `break`.)
9. 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 iterates: $z_{n+1} = z_n^2 + c$ with $z_0 = 0$. Points which do not diverge are within the set; we will define divergence as $|z_n| \leq 2$ for $n \geq 70$.

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 final array with the appropriate index
4. Use a `while` loop to perform the iteration. It is probably easiest to create an array `[x,y]` for z and to evaluate the real and imaginary components separately. 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

5 *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.

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).

```
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

    # 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.

5.1 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 afterthought. 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 9 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.

5.2 *Exercises*

In class

1. Write your own version of the factorial function in the notes (do something differently: use `for`, 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. Write a simple function to split an array into even and odd entries, and return two arrays (use appropriate start and step values)
4. Write a simple function to interleave two arrays, returning one array (it should be the inverse of your split function)

Further work

1. Extend your simple split function to split into `n` arrays
2. Extend your interleave function in the same way
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.

6 *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.

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. 5.

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 create a figure and then populate it with one or more subplots:

- Create the figure: `fig1 = plt.figure()` The figure can now be referenced, and can be displayed again just by giving its name
- Add axes (i.e. graphs): `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)
- 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('')`

```
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")
```

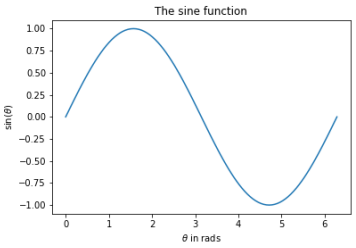

Figure 5: The simple `pyplot` approach and output.

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[5] 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.]

[5] 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.

6.1 *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. Take one of the 2D arrays you generated earlier, or create a new one, and try plotting it with `plt.imshow`. Experiment a little, and see if you can find ways for this visualisation to help you understand array manipulations. (For instance, create a 2D array of ones, and then experiment with setting columns and rows to zero; extend this to play with slices and steps.)

Further work It is often said that sunflower seeds follow a Fibonacci sequence; in this exercise, you will explore what this means.

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 seed number.

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 r and θ
2. Now calculate arrays of the cartesian coordinates in the usual 2D polar way
3. You will now need to plot points (no lines), taking points in steps of f_m (where this is a number in the Fibonacci sequence). Try 5 as the first 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]
4. Do the same thing, but now use `plt.polar(theta,r,'o')` (or whatever style you want). Note that the order of variables is *not* the scientific standard !
5. 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

7 *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.

7.1 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.

7.2 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"

7.3 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

7.4 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 Desktop UCL it is easiest to do the latter. You should edit N:ġitconfig (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 Desktop UCL 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.)

7.5 *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`).

8 *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 Desktop UCL 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.

9 *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/`. The key points can be summarised as follows:

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

Figure 6: Notice how we indent to align the arguments, and use parentheses to implicitly continue a line.

- Four spaces per indentation level (not tabs)
- Lines should not be more than 79 characters
- Split long lines in parentheses, as shown in Fig. 6
- 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)

There is also a style guide for docstrings: `https://www.python.org/dev/peps/pep-0257/`.

10 *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