MATH3511/6111: Scientific Computing
Mathematical Sciences Institute, Australian National University
Semester 1, 2022

Lab 0: Introduction to Python
Lindon Roberts
Due date: 9am, Monday 7 March (week 3)

This lab is based on earlier versions by Stephen Roberts, Graeme Chandler, Jimmy Thomson, Linda Stals and Kenneth Duru.

1 Introduction

These exercises are intended to introduce you to the Python programming language, as well as give programming tips that will help when working through some of the later tutorials. The concepts that will be covered include

- Basic Python
- If/Else statements
- For/While loops
- Functions
- Modules

1.1 Assessment

In each lab, you will be asked some questions. You should provide answers to these questions in your lab book. Marks for these questions will be awarded based on the work presented in your lab book.

Remember this is a mathematics course. The aim is not for you to prove you can produce some sort of output, you must also show that you understand the output. When you have results in your lab book, it is a good idea to include a small discussion about your output: Are the results what you expected? Why or why not? How did you verify your results? What might be some of the reasons why the results are incorrect? During the course we will introduce you to terminology and techniques to help you answer these questions. If you just print the output, you will not be awarded full marks.

Submission Submit your lab book via Wattle. This should include your code, any outputs (e.g. tables of results or figures) and your responses to the questions. I recommend you write a single Python `.py` file which has all your code, and your answers as 'comments': you can print this file to a pdf, and also attach a printout of any outputs (appropriately labelled).

'Comments' are lines of code which are ignored. They are usually used to describe what a piece of code is doing, and are considered an essential part of well-written code. In Python, any line that starts with a # symbol is a comment, as is any part of a line after a # symbol. For example,

```
# This is a comment - you can say anything here
# The next lines do some simple calculations:
1 + 2
5 - 3  # the rest of this line is also a comment
```

Lab 0 Assessment This lab is an introductory exercise which you should complete independently before your first workshop in week 3. It is not graded, but must be completed as a hurdle requirement for the course (i.e. you must submit this lab book to pass the course, but it does not affect your final grade). Your submission needs to be a genuine attempt to complete the exercises.

2 Installing Python

If you have not done so already, you can install Python on your personal computer by downloading the Anaconda distribution (`https://www.anaconda.com/products/individual`). This includes basic Python plus a number of common packages for scientific computing and data science. It also includes Spyder and Jupyter, two programs for writing and running Python code; I recommend that you use Spyder. Anaconda is also available on all ANU computers.

2.1 Spyder

Once you have installed Anaconda, you should be able to run the Spyder program. A screenshot of Spyder is shown in Figure 1.

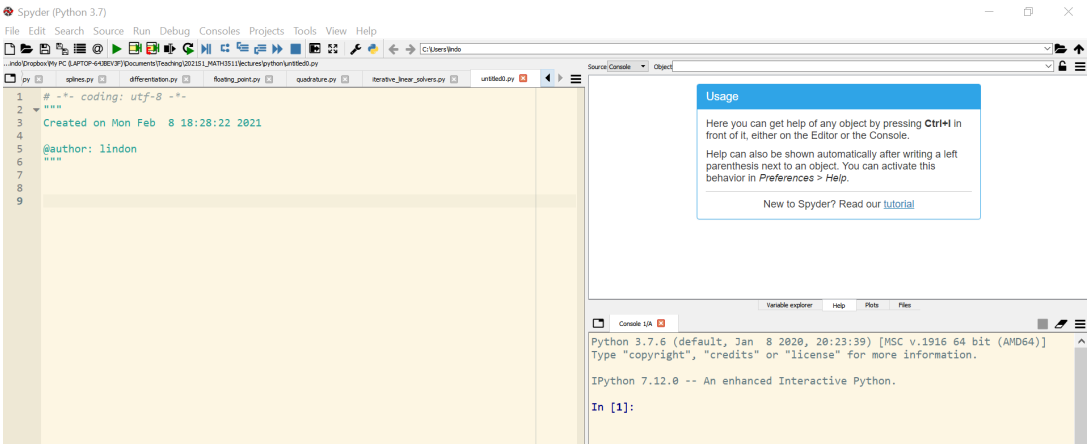

Figure 1. Screenshot from Spyder.

The different parts of the screen are:

- Left: a text editor where you can write your code.
- Top-right: shows various useful things, most importantly any plots you generate (if you click on the 'plots' tab).
- Bottom-right: a Python terminal, where you can run code.

Running Code You can type Python commands directly into the Python terminal window (press 'Enter' to run the commands you have typed). However, for anything beyond 1–2 lines of code, I recommend you write it in the editor, save the file (using the `.py` extension) and run by pressing the "Run code" (green triangle) button in the top menu. This runs all commands in your file.

To run a part of your code, highlight the commands you want to run and press the "Run selection or currrent line" menu button, 3 across from the "Run code" button.

3 Basic Python

From the Python terminal (bottom-right window) you can run simple Python commands. You can use this as a calculator by typing an expression and pressing 'Enter'.

3.1 Calculations

First, let's do some basic calculations in the Python terminal. For example:

```
In [1]: 1 + 1
Out[1]: 2

In [2]: 123 * 456
Out[2]: 56088

In [3]: 2**3
Out[3]: 8

In [4]: 10 - 3
Out[4]: 7

In [5]: 9 / 4
Out[5]: 2.25
```

These commands are addition, multiplication, exponentiation, subtraction and division.

Warning! Note that Python uses `a**b` to mean a^b, not `a^b`.

If we want to represent very large or small numbers, we can use scientific notation: in Python we can represent $a \times 10^b$ as `aeb`. For example

```
In [6]: 1e3 * 2
Out[6]: 2000.0

In [7]: 5.1e-3 + 5.2e-3
Out[7]: 0.0103
```

Python has a small number of other mathematical operations built-in (many more are available in the `math` module which we will cover shortly).

```
In [8]: abs(-3.5)
Out[8]: 3.5

In [9]: max(3, 1)
Out[9]: 3

In [10]: min(3, 1)
Out[10]: 1

In [11]: 7 % 3
Out[11]: 1

In [12]: 7 // 3
Out[12]: 2
```

These calculate the absolute value, maximum and minimum of numbers. The `%` operator is the 'modulus' or remainder operator (`x % y` is the remainder when you divide x by y, and is a number from 0 to $y-1$). The `//` operator performs integer division: `x // y` returns x/y, rounded down to the nearest integer.

You can also assign values to variables using the = sign. For example:

```
In [13]: x = 10

In [14]: x
Out[14]: 10

In [15]: y = 3 * x

In [16]: y
Out[16]: 30

In [17]: x = x + 1

In [18]: x
```

```
Out[18]: 11
```

Note that when we change the value of `x`, the value of `y` is not changed (check this yourself!).

You can check whether certain expressions are true or false using different comparison operators. To test if $x = y$, we use the double-equals operator `==` (since we already use `=` for assigning variables). We also have $<$ (`<`), $\leq$ (`<=`), $>$ (`>`) and $\geq$ (`>=`). To test if things are not equal, $x \neq y$, we use the `!=` operator. For example:

```
In [19]: 1 == 1
Out[19]: True

In [20]: 1 == 3
Out[20]: False

In [21]: 3 * 2 > 5
Out[21]: True

In [22]: -1 <= 0
Out[22]: True

In [23]: 0 < 0
Out[23]: False

In [24]: 1 != 1
Out[24]: False

In [25]: 2 != 1
Out[25]: True
```

3.2 Strings and Printing

In programming, pieces of text are called 'strings'. You create a string by writing your text within single or double quotes.

```
In [26]: myname = "Lindon"

In [27]: myname
Out[27]: 'Lindon'
```

Warning! Python does not treat a string representing a number as an actual number (e.g. `x = "3"` is different to `x = 3`).

In the Python terminal, we see the answer immediately after we perform a calculation. When we write a long piece of code, this is not the case (this is very helpful if we have many calculations!). If you want to check the value of something in the middle of some code, you can ask Python to print it.

```
In [28]: print(myname)
Lindon
```

Note that the result does not have `Out[28]:` it is being printed to the terminal window, not returned as an output of the calculation/command. The `print` command also works with numbers, or any other Python variable.

3.3 Conditional Statements

Sometimes we want to do something conditionally (only under certain circumstances). The `if` statement allows us to do this. In the text editor (Spyder left-hand window), type the following code, and press the "Run code" button to execute it.

```
x = 10
if x < 50:
    print("x is less than 50")
```

Notice that the line after the `if` statement is indented. Python uses white space to organise code, so you must be careful to use whitespace correctly. You can use tabs or spaces for whitespace, but you must be consistent or problems will occur; the Python recommendation is to always use 4 spaces for indentation.

Fortunately, Spyder automatically adds 4 spaces after conditional statements (and loops, see below). If you press the 'tab' key, Spyder will indent the current line by 4 spaces (and 'Shift + tab' removes 4 spaces).

If you want to test multiple conditions, use the 'elif' (short for "else if") and 'else' statements:

```
if x < 0:
    print("x is less than 0")
elif x < 50:
    print("x is less than 50 and bigger than 0")
else:
    print("x is greater than or equal to 50")
```

Lastly, if you want to test multiple conditions, you can use the 'and' and 'or' keywords:

```
if x < 0 and x % 2 == 0:
    print("x is negative and even")

if x < 0 or x % 2 == 0:
    print("x is either negative or even (or both)")
```

3.4 Loops

The other key programming task is to execute commands multiple times. We can do this with loops. There are two types of loop: 'for' loops and 'while' loops.

'For' loops look like this:

```
for i in range(10):
    print(i)
```

and 'while' loops look like this:

```
i = 0
while i < 10:
    print(i)
    i += 1  # short for "i = i + 1"
```

Both of these codes produce the same output.

Warning! If you forget the `i += 1` statement in a 'while' loop, your code will run forever. You can stop a code by clicking on the Python terminal and pressing 'Control + c'. I mostly use 'for' loops, which will never run forever.

The `range` function, as the name suggests, is used to define ranges of numbers:

- `range(n)` loops over $0, 1, \ldots, n-1$ (inclusive), in that order.
- `range(a, b)` loops over $a, a+1, \ldots, b-1$ (inclusive), in that order.
- `range(a, b, s)` loops over $a, a+s, a+2s, \ldots, b-1$ (inclusive), in that order. If $b-1$ is not in this sequence, it stops at the largest value less than $b-1$. You can loop backwards by using $s < 0$, but of course then you need $a > b$.

`range` behaves like a list (see below), but is a special Python object for writing loops. You can turn it into a list with

```
mylist = list(range(10))
print(mylist)
```

You can find out more about a function using the `help` command:

```
help(range)
```

Lab Book 1. Write code which calculates the sum $2 + 4 + 6 + \cdots + 2n$ using a 'for' loop (where the value of n is saved in a variable `n`. Check your code by comparing with the true value $n(n + 1)$.

3.5 Functions

To define a function, you can use the 'def' keyword (short for 'define'). The inputs to the function are written in brackets, and the output is given by a `return` statement.

```
def f(x):
    return 2 * x
```

This code returns a value that is double the input value.

The function can then be evaluated:

```
# Just print the result directly
print(f(2))
# Save the result to a variable
input_x_value = -5.8
y = f(input_x_value)
print(y)
```

It is always a good idea to add comments to your code to describe what it's doing (and why). This allows other people (and yourself later on!!) to understand what the code is doing, and to modify or fix it. Python has a built-in way for you to describe your function, called a 'docstring'.

```
def ex(x):
    """
    Calculate the exponential e^x using a Taylor series.

    This is a docstring: it can have as much information as you want.
    Usually, this means at least explaining what the function does,
    what input(s) it expects, and what output(s) it has.

    Any limitations/restrictions on inputs should go here (for example,
    'this only works if x > 0').
    """
    ans = 0
    n = 1
    term = 1
    while ans + term != ans:
        ans = ans + term
        term = term * (x / n)
        n = n + 1
    return ans
```

If you comment your function using a docstring, it works with the `help` command:

```
help(ex)
```

Lab Book 2. Using this program, what is the approximation to e^1, e^{10} and e^{-10}. What are some techniques you might use to check the output?

Lab Book 3. The Taylor series expansion of exp is

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

When writing code to implement Taylor series expansions we can't sum to ∞. Explain what has been done in the above code to get around this problem.

3.6 Lists

Often we want to look at collections of values. The standard way to do this in Python is with a list:

```
mylist = [1, 4, 9, 16, 25, 36, 49]
```

You can refer to an individual entry in a list based on its index:

```
print(mylist[1])
```

Warning! The first item in a list has index 0 (`mylist[0]`). So, if a list has n elements, the last item in the list has index $n-1$ (`mylist[n-1]`).

We can write a 'for' loop over all elements in a list:

```
for x in mylist:
    print(x)
```

and check how many elements are in a list using `len`:

```
print(len(mylist))
```

Alternatively, we can loop over all indices:

```
for i in range(len(x)):  # loop over i=0,...,n-1 where the list x has n elements
    xi = x[i]  # i-th element of list x
    print(i, xi)
```

We can also pick out subsets of a list using 'slicing':

```
print(mylist[2:5])
print(mylist[3:])
print(mylist[:-2])
```

Negative indices refer to the end of the list (e.g. `-1` is the last element of a list). Slicing `mylist[a:b]` goes from index `a` to index `b-1` (just like with `range`!), where leaving `a` or `b` blank means the start/end of the list.

Lab Book 4. What slice returns `mylist` without the first and last elements (i.e. the slice is `[4, 9, 16, 25, 36]`)?

Vectors We can use lists like vectors in $\mathbb{R}^n$. However, the Python package NumPy has specialised functions for creating and manipulating vectors and matrices (and can do lots of common linear algebra tasks). We will cover NumPy in much more detail in Lab 1.

4 Modules

To make Python more useful than a simple calculator, and to avoid rewriting common functions yourself, you use code other people have written. These functions are grouped together into packages, called 'modules'. For example, the `math` module (which is part of standard Python) has many standard mathematical functions. To use functions in a module, we have to import them by name:

```
from math import sqrt, sin, pi
print(sqrt(2.25))
print(sin(pi/2))
```

Often, importing each function one-by-one is inconvenient (e.g. if we are using many functions from a module). An alternative is just to import the whole module, then you can refer to any function:

```
import math
print(math.sqrt(2.25))
print(math.sin(math.pi/2))
```

To avoid lots of typing, we can assign a module a shortcut name:

```
import math as m
print(m.sqrt(2.25))
print(m.sin(m.pi/2))
```

It is good Python practice to put all `import` lines at the top of your file. A full list of functions in the `math` module is available online: `https://docs.python.org/3/library/math.html`.

Lab Book 5. How accurate is the `ex` function defined earlier compared to the `math.exp` function? Try evaluating `ex(x)` and `exp(x)` for values of `x` from -20 to 20. We will study these differences later in the section on floating point numbers.

Optional arguments When using `help` to view information about a function, sometimes inputs are written in square brackets. For example,

```
import math
help(math.log)
```

prints:

```
Help on built-in function log in module math:

log(...)
    log(x, [base=math.e])
    Return the logarithm of x to the given base.

    If the base not specified, returns the natural logarithm (base e) of x.
```

This says that we can compute $\log_e(x)$ with `math.log(x)` and $\log_2(x)$ with `math.log(x, base=2)`.

4.1 Directory Paths

If you import a module, Python will look for it in two locations:

- A Python module included as part of your Python installation. Python has many built-in modules (like `math`), and Anaconda adds many more (like `numpy` for linear algebra). If you find other modules, you can usually install them by running `pip install package_name` in the terminal window.
- A `.py` file with the same name as the module, in the same directory as your current code. This is useful if you have written functions you want to re-use for other tasks.

5 Exercise: Black-Scholes Formula

The Black-Scholes formula is a famous equation in mathematical finance (Fischer Black & Myron Scholes, 1973). The underlying mathematical theory was awarded the Nobel Prize in Economics in 1997.

A 'European call option' is a type of financial contract which gives someone the ability to buy one share in a company for a fixed price ("strike price") on a fixed date (the "maturity" date). If the strike price is less than the actual share price on that date, then they will generally use this ability, since they make a profit. If the strike price is more than the actual price, they will not use this ability, and they make no money. The Black-Scholes formula is a way of calculating the value of this deal before the maturity date.

The formula for the price of the deal, C, is:

$$C = \Phi(d_1)S - \Phi(d_2)Ke^{-rT},$$

where

$$\begin{aligned}\Phi(x) &= \frac{1 + \operatorname{erf}(x/\sqrt{2})}{2},\\ d_1 &= \frac{\ln(S/K) + (r + \sigma^2/2)T}{\sigma\sqrt{T}},\\ d_2 &= d_1 - \sigma\sqrt{T}.\end{aligned}$$

Note that erf in the function Φ is called the "error function".[1] The variables are:

- K, the strike price (in \$).
- S, the current share price (in \$).
- T, the time before maturity (in years).
- r, the current interest rate (e.g. for deposits in bank accounts).
- σ, the "volatility" of the share price (roughly, this is the standard deviation of changes in the share price: larger σ means the share price is more variable day-to-day).

Lab Book 6. Look through the online documentation of Python's `math` module. What function can you use to evaluate $\operatorname{erf}(x)$? What is the mathematical definition of $\operatorname{erf}(x)$ (in terms of a single integral)?

Lab Book 7. Write a function `phi` which calculates $\Phi(x)$. Then, write a function `european_call` which takes in the inputs K, S, T, r and σ, and returns the price C. What is the price of a European call option with $K = 90$, $S = 100$, $r = 0.03$, $T = 0.5$ and $\sigma = 0.1$? Does the price C increase or decrease as σ increases?

[1] The function $\Phi(x)$ is the cumulative distribution function for a standard normal random variable.

6 Exercise: Electric Circuit

Problem Figure 2 shows a voltage source $V = 120\text{V}$ with and internal resistance R_s of 50Ω supplying a load of resistance R_L. Find the value of load resistance R_L that will result in the maximum possible power being supplied by the source load. How much power will be supplied in this case?

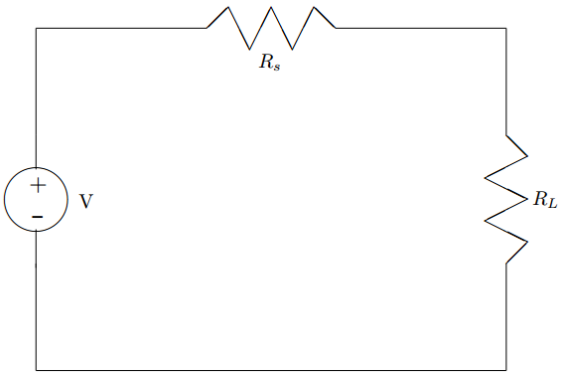

Figure 2. Voltage source with voltage V and an internal resistance R_s supplying a load with resistance R_L.

Mathematical Description In this program we need to vary the load resistance R_L and compute the power supplied to the load at each value of R_L. The power supplied to the load resistance is given by the equation

$$P_L = I^2 R_L,$$

where I is the current supplied to the load. The current can be calculated by Ohm's law:

$$I = \frac{V}{R_s + R_L}.$$

Pseudocode It is important to always break up our problems down into smaller sections. It is easier to understand, debug and check smaller sections of code. This can be done using pseudocode:

Algorithm 1 Power Supply.

1: Build a list of possible values for the load resistance R_L. The list will vary R_L from 1Ω to 100Ω in 1Ω steps. Calculate the current for each value.
2: Calculate the power P_L supplied to the load for each value of R_L.
3: Plot the power supplied to the load for each value of R_L and determine the value of load resistance resulting in the maximum power.

Python Code Let's first write a function that calculated the current using Ohm's law.

```
def current(RL,V,RS):
    """
    Calculate the current given the voltage source with voltage V
    and internal resistance RS supplying a load of resistance RL.

    Input: RL = load of resistance (list of floating-point numbers)
         : V = Voltage (floating-point number)
         : RS = internal resistance (floating-point number)

    Output: current (list of floating-point numbers)
    """
```

```
    n = len(RL)
    I = n*[0.0]  # I is a list with the same size as RL
    for j in range(n):
        I[j] = V / (RL[j] + RS)
    return I
```

Warning! The use of the 'for' loop in Python code like this is usually very slow and is considered to be poor programming practice. Later, we will look at more efficient ways to write such codes.

We can now test the function by building an example list `RL`:

```
RL = list(range(1,6))
I = current(RL, 1.0, 0.0)
print(I)
```

Note that, in Python, if we want to indicate that a number can take non-integer values[2], we usually write it with a decimal point (e.g. `1.0` instead of `1`). Writing a number without a decimal point indicates it represents an integer.

Lab Book 8. Is the output produced by the above code correct? Why or why not? If it is not correct, what modifications must be made to the code to correct it?

Lab Book 9. Write a function `power` which takes in a list of currents `I` and resistance loads `RL`, and returns a list of values of P_L given by the formula above. Test your `power` function by running:

```
RL = list(range(1, 6))
I = current(RL, 1.0, 0.0)
PL = power(I, RL)
print(PL)
```

Now, we can write code to plot a graph of P_L versus R_L. We will use the `matplotlib` package, which is part of Anaconda. The below function produces a plot of R_L versus P_L: do not worry about the details, we will cover plotting in more detail in a later lab.

```
import matplotlib.pyplot as plt

def power_plot(RL, PL):
    """
    Plot the power PL versus the load of resistance RL.

    Input: RL = load of resistance (list)
         : PL = power (list)

    Output: none
    """
    plt.clf()
    plt.plot(RL, PL)
    plt.title('Plot of power versus load resistance')
    plt.xlabel('Load resistance (Ohms)')
    plt.ylabel('Power (Watts)')
    plt.grid()
    plt.show()
    return
```

Finally, we can answer the original question with the following code:

[2] On computers, values in $\mathbb{R}$ are represented as 'floating-point numbers'. We will discuss these in detail later in the course.

```
# Set the internal resistance to 50 Ohms
RS = 50.0
# Set the voltage source to 120V
V = 120.0
# Create a list of possible values for the load resistance
RL = list(range(1, 101))
# Calculate the current and power
I = current(RL, V, RS)
PL = power(I, RL)
# Plot the power versus load resistance
power_plot(RL, PL)
```

The resulting plot should suggest that the maximum power is supplied to the load when the load's resistance is 50Ω.

Lab Book 10. By using list indexing/slicing, find the value of P_L corresponding to this optimal value of R_L.

7 Examples of Good Programming Practice

The following gives some examples of what is considered to be good programming practice. They are very useful habits up and will save you a lot of time and frustration, especially when we look at more complicated programs.

- **Use functions:** This helps to break the problem down into smaller sections. It is easier to understand and debug a small section of code, and it also allows you to reuse code rather than rewriting it. If each function is narrowly focused and responsible for one particular aspect of your code, you are less likely to get a cascade of errors, which is very difficult to debug.
- **Rerun your code often:** When learning a new language you may want to run your code every time you make a change (I do). That way you can easily track down the source of any bugs. Even when you become familiar with the language, it pays to run your code regularly. It helps to narrow down the parts of the code you have to look at when you get errors.
- **Use the mathematics:** Even if you don't know what the exact solution is, the mathematics will often tell you about some of the properties of the solution. For example, if you get a negative answer when calculating the square root of a real number then something has gone wrong.
- **Write a pseudocode first:** This allows you to focus on the logic of the problem. It also helps you structure the flow of information. That is, what do you need to pass into a function and what information do you need to get out it.
- **Format your code:** You should use spaces, comments, meaningful variable names, etc. to make the code easier for a human to read. It will save you a lot of time if you want to ask someone else for help. It also forces you to better structure the code.