The

Survival Guide

for

3rd Year Computational Physics Labs

Authors:

Yi Shuen C. Lee

Daniel Ferlewicz

Part 3 Laboratories, School of Physics

The University of Melbourne

October 10, 2025

Contents

1 Introduction

Welcome to 3rd Year Laboratory and Computational Physics. This document contains a few important things that will get you set up for the computational labs including the basics for using the terminal, how to get setup remotely or locally with university computers, an introduction to python and tips on how to write a good program, and an example of how to run a computational lab with report writing.

1.1 Log-book: Writing as you go

Now before we dive into the more technical side of things, we would like to remind you of a few simple, but very important things.

In third year labs, we ask you to provide a log-style report at the end of the fortnight. Your report should be chronological and rigorous with respect to the physics (careful analysis including errors, clear explanations of motivations and understandings) and most importantly, you should be **writing as you go**. We want your report to reflect your work progress throughout the two weeks, whether they are right or wrong. If you made a mistake and realised it later, **do not rip out or delete what you have written**. Instead, we suggest that you make a retroactive comment on what you think you've done wrong and how you've managed to fix the error. That way, you demonstrate that you are able to present you thought process and demonstrators marking your books will know where you've spent your time.

Those of you who have done the second year physics laboratory subject should already be familiar with the basic requirements – in third year we have similar expectations, except that each lab stretches across a fortnight and hence you should have a lot more procedures/results to discuss. In summary, we expect your log-book to to fulfill two aims:

1. To act as a true scientific log. Anyone should be able to take your log book into the lab and replicate your experiment without needing to refer to other materials.
2. The second aim is pedagogical. We need to find out how much you understand. For this reason, and because being able to convey results clearly is an important skill in science, your report needs to have some basic structure to it, showing off your analysis skills and critical thinking.

The general recommended structure is: Background theory, Aim, Method, Result, Analysis/Discussion and Conclusion. Nevertheless there are no strict rules on how you should structure your labs, as long as it makes sense and is easy to follow. Regardless of what your findings are, you must **always include a conclusion** at the end of your log-book to summarise what you have learned and/or discuss what you think have gone wrong that you did not manage to obtain desirable results. An "inconclusive conclusion" is better than none (seriously, we insist on it).

There will be questions asked of you in your lab notes. Some of them are mandatory in the sense that if you don't answer them, you won't have the information to proceed with your labs. Some of them are there to act as a guide to help you understand a certain concepts better. How you include your answers to these questions in your log-book is entirely up to you.

1.2 Mark distribution: Computational Lab

This marking rubric acts as a guideline to what we expect from your log-books. Again, there are no strict rules on how you structure your logs!

(a) **Introduction: 4 marks**

- Set the scene for your computational lab.
- State the problem you are aiming to solve in your own words.
- Include any preliminary theory or information that will help with writing your code.

(b) **Procedure: 6 marks**

- Record what you do when you do it.
- Use simple diagrams to help when needed.

- Should be considered to be read with the experimental notes alongside - so you should not copy large sections.
- Keep it informal and record as you go.

(c) **Results/Code: 6 marks**

- Not just about 'getting the answer' but about having some logically reasonable code.
- Submitted code should compile (where applicable) and run.

(d) **Interpretation and Analysis: 6 marks**

- Give meaning to your algorithms and the output of your programs.
- Speculate on problems or improvements when results differ from expectations.
- Use questions in the lab notes as a guide.

(e) **Conclusion: 2 marks**

- Wrap up the exercise.
- Quote any final numeric results where applicable.
- Discuss what can be concluded from your work.

(f) **Record keeping: 6 marks**

- Write your logbook during the lab class.
- You should not need to do any work outside the lab if you are recording what you do when you do it!

(g) **Participation: 4 marks**

- Attend your labs.
- Actively contribute to group discussions with your peers and instructors.
- Work as a group and use available resources to solve problems.

2 Terminal 101

First, what is a terminal? It's a very simple but very powerful way of using a computer. Here we will cover how to do things that you would be used to doing with the file explorer of your modern-day computer.

2.1 Basic terminal commands

- **Changing directories**
  `cd <path/to/directory>`
- **Listing all files in the current directory**
  `ls`
- **Copying a file into another directory**
  `cp <filename> <path/to/newfilename>`
- **Moving a file into another directory**
  `mv <old/path/to/filename> <new/path/to/filename>`
- **Renaming a file**
  `mv <oldfilename> <newfilename>`
- **Making a new directory**
  `mkdir <path/to/new/directory>`
- **Removing a file**
  `rm <filename>`
  CAUTION: By using these `rm` command, the files/directories gets removed permanently and cannot be recovered.

2.2 Unzipping compressed files

1. Open a terminal window
2. cd to the directory with the folder you want to unzip
3. Type in the following command:
   `tar -xvf <file_name.tar.gz>`

EXERCISE 1: Ensure that you are ssh'd into the Baker lab computers. Copy over "debugging_exercises.tar.gz" from Physics Common. Unpack and rename the unpacked directory to whatever you like. (Path to Physics Common: `/home/unimelb.edu.au/PhysicsCommon/3rd_year`)

3 Setting up

The computers in the Baker lab and 3rd year labs have a special build on them that will let you log in with your unimelb username and password, with access to a number of different software packages. These are uniform across each machine and are therefore the best place to work on and execute your code so that you can avoid any issues with different versions of software.

3.1 JupyterLab

JupyterLab is where you will be running your scripts (e.g. the `.py` and `.ipynb` files). To launch JupyterLab, you simply need to open a terminal window and type the command `jupyter-lab`. If a browser window does not pop up automatically, there should be some links in your terminal window. You can copy any one of these links into a browser tab, and that should work.

If the above fails, please ask your demonstrators for help - **Do not attempt the following yourself steps yourself.**

1. For demonstrators, try: `pip install jupyterlab`

2. After it finished installing, and every time after that, enter the following command, where YYY is the same three-digit number from the remote connection steps:

   ```
   jupyter-lab --port 8YYY --no-browser
   ```

   If this does not run and the installation ended with some yellow text with warnings, do the following:

   (a) In the terminal, run
   ```
   emacs ~/.bashrc
   ```
   Or use vim, or any other text editor

   (b) Go to the end of the file (looking at the top panel), taking care to not add/delete anything else, and add the line
   ```
   export PATH="/home/student.unimelb.edu.au/username/.local/bin:$PATH"
   ```

   (c) Save and close the file

(d) Open a new terminal tab and try the jupyter-lab command again

(e) You may have an error involving unicode. If so, run the command
`pip install markupsafe==2.0.1`

(f) You should now restart the terminal, and run
`pip install jupyterlab --force-reinstall`

3. You will see it print a link inside of a bunch of text. Copy this link into any web-browser and press enter.

4. You can then open a terminal to run programs, and use the file browser on the left to edit.

EXERCISE 2: Set up JupyterLab and open whatever program (i.e. '.py') file you want and pull up a terminal window side-by-side to the file you just opened - this is just to show you what JupyterLab can do.

3.2 Setting up conda environments (for EPP labs)

See notes for the CPV lab and Higgs lab.

4 Introduction to Python

4.1 Importing Libraries

In order to use functions in a python module, your program will need to be able to access the module to begin with. We do this using the `import` statement. There are three main things you should know about the `import` statement:

(a) Importing the whole library → `import math`

(b) Importing part of a library e.g. the `stats` sub-package from the `scipy` library → `from scipy import stats`

(c) Importing a library using an alias → `import numpy as np`
By doing this, you can use functions from `numpy` by writing `np.<function>(args)`

You probably won't need this for this subject but we will mention it, just in case!
- You can also install extra modules/libraries which are not already on your machine using the command below in your terminal window

Code start

```
python3 -m pip install <module name>
```

Code end

4.2 General Python stuff

(a) **Lists**

- A list is a sequence of data values of any type (i.e. it can be a mix of integers, strings, floats, lists, arrays etc.)
- Indicated by enclosing its elements in a pair of square brackets, [ ] Values in a list is mutable i.e. can be changed and/or replaced

(b) **Tuples**

- Tuples are pretty much like lists
- BUT the elements are enclosed in a pair of parantheses, ().
- More importantly, tuples are immutable i.e values in a tuple cannot be changed and/or replaced.

(c) **`range()` function**

- Used to generate a sequence of numbers.
- Has 3 arguments, two of which are optional
- The first argument (optional) is the starting number
- The second (required) is the end number
- The third (optional) is the *incrementor* or *decrementor*
- Altogether: `range(start, stop, step)`
- The sequencing starts from 0 (by default), and increments by 1 (by default), and stops before a specified number.
- **Note:** the specified 'end number" is excluded from the sequencing.
  e.g. `range(5)` gives you a sequence of numbers from 0 to 4.

- If you try to print the range function itself i.e. `print(range(5))` you won't get anything.
- Instead you can use it to create a list of numbers - see following examples:
  * list(range(3)) → [0, 1, 2]
  * list(range(2,5)) → [2, 3, 4]
  * list(range(3,8,2)) → [3, 5, 7]
  * list(range(10,0,-1)) → [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
  * list(range(10,0,2)) → [] i.e. empty list!

(d) **Python indexing**

- In python, the first index is always zero
- In other words, the n^{th} element in the list has an index of $n-1$.
- For example, given a list `List=[9, 5.3, "Hello", (1,2)]`:
  The string "Hello" is the third element in the list and hence takes the index of 2.
- The last element of the list takes the index -1, the second last element has index -2 and so on

(e) **Slicing a list or tuple**

- You can access a selected range of items in a list or tuple by using the slicing operator colon, :
- Syntax:
  `sliced_list = original_list[InitialIndex : EndIndex : IndexJump]`
- The slice starts at `InitialIndex` but finishes one before the `EndIndex`
- Here are a few examples on slicing:
  * `Example = [1, 2, 3, 4, 5, 6, 7, 8, 9]`
  * `Example[:]` or `Example[::]` returns all elements of the original list
  * `Example [2:5]` → `[3,4,5]`
  * `Example[3:9:2]` → `[4, 6, 8]`
  * `Example[::2]` → `[1, 3, 5, 7, 9]`

(f) **Replacing elements in a List** (an example)

Code start

```
>>> example = [1, 2, 3, 4]
>>> example[-1] = 1
>>> example
[1, 2, 3, 1]
```

Code end

(g) **String formatting**

- There are a number of ways to format python strings, but the easiest is the fstring. You can do variable manipulation inside the curly brackets too:

  Code start

  ```
  >>> name = "Alex"
  >>> print(f"Hello, my name is {name}")
  "Hello, my name is Alex"
  >>> x = 3
  >>> print(f"{x} squared is equal to {x**2}")
  "3 squared is equal to 9"
  ```

  Code end

- See more, like setting the number of significant figures in: https://zetcode.com/python/fstring/

4.3 Selection Statements: `if, elif, else`

Selection statements allow a computer to make choices based on given condition(s). A common example is the **if-else** statement which is used to create a decision structure and hence allows a program to have more than one path of execution.

The syntax of an **if-else** statement is:

Code start

```
if <condition>:
    <sequence of statements-1>
else:
```

```
    <sequence of statements-2>
```

Code end

Note that the condition in any `if-else` statement must be a Boolean expression i.e. an expression that evaluates to either **True** or **False**. An example of an `if-else` statement is shown below:

Code start

```
first = 5
second = 9
if first > second:
    maximum = first
    minimum = second
else:
    maximum = second
    minimum = first
```

Code end

In some cases, you might have multiple conditions, each of which requires its individual path of execution. That is where the `elif` statement steps in. The syntax of an `if-elif` statement is:

Code start

```
if <condition-1>:
    <sequence of statements-1>
elif <condition-2>:
    <sequence of statements-2>
elif <condition-3>:
    <sequence of statements-3>
else:
    <sequence of statements-4>
```

Code end

NOTE: The `else` statement at the end is optional - if you do not choose use this statement, the code will simply not perform any sort of execution if neither condition-1, 2 nor 3 are satisfied. So it depends on what you want to do with your program.

4.4 Loops in Python

The Python programming language provides the following types of loops to handle looping requirement:

1. **for loop**: Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
2. **while loop**: Repeats a statement or group of statements while a given condition is **True**. It tests the condition before executing the loop body.
3. **nested loops**: can use one or more loop inside any another e.g. **while** within a **for** loop.

4.5 The `while` loop

The **while**-loop can be used to describe *conditional iteration*. A conditional iteration requires a *continuation condition*, i.e. a condition be tested within loop to determine if it should continue. For example: A program's input loop that accepts values until user enters a 'sentinel' that terminates the input. A `while`-loop is also sometimes called entry-control loop since the given condition is tested at top of loop so that statements within the loop can execute zero times if the condition fails, and one or more time(s) if the condition(s) are satisfied.

The general syntax for `while`-loop is:

Code start

```
while <condition>:
    <sequence of statements>
```

Code end

Caution: improper use of the continuation condition will potentially lead to an *infinite loop*. Here is an example on how to use the `while`-loop:

Code start

```
target = 100
Sum = 0
while Sum < target:
    Sum+=1
```

```
print(Sum)
```

Code end

where the print-statement will output “100". Basically, the `while`-loop terminates (i.e. the summation stops) when `Sum<target` is no longer true.

4.6 The for loop

A **for**-loop is a repetition of statements. Each repetition of action is known an *iteration*. The **for**-loop is a control statement that supports a definite iteration (i.e. it stops at some point). The syntax of a **for**-loop is given by:

Code start

```
for <iterating_> in <sequence>:
    block of statement(s)
```

Code end

Two examples of **for**-loops are shown below, one is a stand-alone **for**-loop and the other is a **for**-loop with a nested **if-else** statement:

Code start

```
Example 1:
listofitems = [1, 2, 3, 4, 5]
for num in listofitems:
    print(num)

Example 2:
for val in range(0,7):
    if val <=3:
        print("Small number")
    else:
        print("Large number")
```

Code end

Something you may find useful is using for-loops in combinations with defining a list. You could do something like

Code start

```
numbers = [i for i in range(5)]
print(numbers)
```

Code end

As much as `for`-loops are great for repetitive statements, we strongly advise that you attempt to use `numpy` arrays for large iterations involving calculations, as they are **MUCH** more efficient (i.e. faster) - see Section 4.9 on `numpy` arrays.

4.7 Python functions

A function is a group of statements that exist within a program for the purpose of performing a task. Much like in mathematics, functions take a set of input values, perform some calculation based on them, and optionally return a value.

4.7.1 Steps to creating a Function

- Defining a function
- Write the statements within a function to perform a task
- Call the function
- Return the value (if any)

4.7.2 Defining a Function

The following shows the syntax of defining a function:

Code start

```
def <function_NAME>(tuple_of_arguments):
    statement
    statement
    statement
```

Code end

and here is an example of how a function can be defined:

Code start

```
def message():
    print(‘this is a simple case’)
```

Code end

4.7.3 Calling a Function

To execute a function, you must call the function: `<function_name> ()`, e.g. if we want to call the function defined in the example above, we need to write a line in the code: `message()`.

4.7.4 Functions: main() and other functions

The **main()** function is typically used to execute other functions and statements all at one shot. For example:

Code start

```
def message1():
    print("Hello World!")

def message2():
    print("Hello again!")

def main():
    message1()
    message2()

main() #this line executes the main() function by calling it
```

Code end

4.7.5 Scope and Local Variables

A variable's *scope* is the part of a program in which the variable may be accessed. *Local variable* is a variable created inside a function and cannot be accessed by statements that are outside the function. Here is an example on how local variables appear in functions:

Code start

```
def main():
    melbourne()
    canberra()

def melbourne():
    birds = 1000
    print("melbourne has", birds, "birds")

def canberra():
    birds = 870
    print("canberra has", birds, "birds")
```

Code end

Notice that different functions within a program can have same variable names. This is because the other functions cannot see or use each others local variables. For example, the code below will result in an error:

Code start

```
def main():
    get_name()
    print('hello', name) #causes an error

def get_name():
    name = input('enter your name:')

main()
```

Code end

because the main() function cannot access the variable 'name' from the get_name() function, i.e. the scope of 'name' is only within the get_name() function

However, just like normal variables, a local variable cannot be accessed by code that appears inside a function at a point before the variable has been created. For example:

Code start

```
def bad_function():
    print("the value is", val) # causes an error
    val = 99

bad_function()
```

Code end

will result in an error because the variable 'val' is called before it is defined.

4.7.6 Return Statement

A value-returning function has a *return statement* that returns a value back to the part of the program that called it. The syntax is:

Code start

```
def <function_name>():
    statement
    statement
    return <expression>
```

Code end

The value of the expression that follows the "return" statement will be sent back to part of the program that called this function. This can be any value, expression, or variable that has a value. A return statement can also send back strings, boolean values and multiple values.

Here is an example on how to use the return statement:

Code start

```
def getFarenheit(C):
    return 9*C/5 + 32

def main():
    celcius = int(input(‘enter a value in celcius:’))
    f = getFarenheit(celcius)
    print(f)
```

Code end

4.7.7 Passing arguments to functions

At times, we will need to harvest and employ information between functions. This example shows how we pass a value from a variable in the `main()` function, into the argument of another function:

Code start

```
def main():
    value = int(input(‘enter a number:’)) #User inputs a number
    doubling(value)

def doubling(number):
    return (number * 2)
```

Code end

4.7.8 Keyword Arguments (kwargs)

While generally, arguments are passed by position to parameter variables in functions, you can also specify which parameter variable the argument should be passed to i.e. a *keyword argument*. For example:

Code start

```
def show_interest(principal, rate, period):
    interest = principal * rate * period
    print(‘The simple interest will be $%.2f’ %interest)
    print(‘The principal is %.2f’ %principal)
    print("The period is %.2f" %period)
```

```
def main():
    show_interest(rate = 0.01, period = 10, principal = 100000.0)

main()
```

Code end

You can also **mix keyword arguments** with positional arguments. However, positional arguments must come first, followed by keyword arguments. For example, with the same `show_interest()` function defined above, we can try:

Code start

```
>>> def main():
       show_interest(10000.0, period = 20, rate = 0.01)

>>> main()
The simple interest will be $2000.00
The principal is 10000.00
The period is 20.00
```

Code end

An error, however, will arise if we put positional arguments after keyword arguments:

Code start

```
>>> def main():
       show_interest(period = 20, 10000.0, rate = 0.01)

SyntaxError: positional argument follows keyword argument

>>> def main():
       show_interest(period = 20, rate = 0.01, 10000)

SyntaxError: invalid syntax
```

Code end

4.8 Object oriented programming: Python "Classes"

Python is an object oriented programming language. An object has its own defined properties and execution methods. A “class" is pretty much a structured object in python which contains functions that are systematically grouped.

We think the easiest way to introduce this concept is by first showing you what they are - so here is an example of a class named ‘Person’:

Code start

```
class Person:
   def __init__(self, name, age)
       self.name = name
       self.age = age

   def greeting(self):
       print(f"Hi, my name is {self.name}"). I’m {self.age} years old.")

   def birthday(self):
       self.age+=1

person1 = Person(name="John", age=36)
person1.greeting()
person1_name, person1_age = (person1.name, person1.age)
```

Code end

Now let’s discuss what these all are:

(a) `__init__` is a special ‘initialise’ function used to assign values to the `class` properties. In the example above, our `class` is a person who has properties "name" and "age".

    Note: The `__init__` function is called automatically every time the class is being used to create a new object.

(b) A `class` is an object in Python, and so it can contain methods i.e. functions that belong to this object/class. In the example above, `greeting()` is a function which executes on objects created using the Person class.

(c) The `self` parameter is a reference to the current instance of the class. It is used to access variables that belong to the class.

(d) The third last line shows how to create an object using the Person i.e. "person1" now holds an instance of the class Person.

(e) The second line shows how to execute a class method i.e. the `greeting() function` on the object "person1".

(f) The last line shows how to access variables in the class from which you've created your object.

4.8.1 Inheritance

In object oriented programming, inheritance allows us to define a class that inherits all the methods and properties from another class. There are two terms we need to know: (i) Parent Class: the class with properties being inherited from and (ii) Child class: the class that inherits from the parent class. Here's an example:

`Code start`

```
 class Person:
    def __init__(self, fname, lname)
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)

class Student(Person):
    pass

x = Student("John", "Smith")
x.printname()
```

`Code end`

In the example above, the Student class is used to create an object. But as you can see from the way the Student class is defined, it inherits the properties and methods from the Person class. Therefore the `printname()` function (a method) from the Person class can be executed on object "x" which is created using the Student class.

4.8.2 Abstract Classes

An abstract class cannot be instantiated on its own. Instead, you will have to create classes that inherit properties and method from an abstract class. To define an abstract class, you need to use the abc module which provides you with the required infrastructures for define abstract base classes.

The following example demonstrates a program that calculates the salary of employees based on their contracts (e.g. part-time, full-time, casual):

Code start

```
from abc import ABC, abstractmethod

class Employee(ABC):

   def __init__(self, first_name, last_name):
       self.first_name = first_name
       self.last_name = last_name

   @property
   def full_name(self):
       return f"self.first_name self.last_name"

   @abstractmethod
   def salary(self):
```

Code end

For different employment contracts, the salary is calculated using different methods. So we first develop an abstract class named Employee, which has a property that returns the full name of the employee. On top of that, the Employee class must have a method that calculates the salary - but we do not know what the method is unless we know what contract the employee is on. Hence we use `@abstractmethod` - a decorator for defining an abstract method/class, to say that this `salary()` property must exist but will be inherited from another object. Now we define another class called `CasualEmployee` which inherits properties from the `Employee` class. The `CasualEmployee` class will also provide the implementation for the `salary()` method:

Code start

```
class CasualEmployee(Employee):
    def __init__(self, first_name, last_name, worked_hours, rate):
        super().__init__(first_name, last_name)
        self.worked_hours = worked_hours
        self.rate = rate

    def salary(self):
        return self.worked_hours * self.rate
```

Code end

4.9 Numpy arrays

Numpy arrays will be your most valuable tool for all computational labs (or in fact for all python programs you write in the future). ALWAYS ALWAYS try to use `numpy` arrays before resorting `for`-loops and if statements. As mentioned before, this will substantially improve the efficiency of your code - or if it is more convincing, it will give you a better mark!

In this section, we will show you some basics on what you can do with `numpy` arrays . It should give you a gist of just how powerful this tool is - but of course we will not cover everything about `numpy` arrays. The first thing you need to do to be able to use `numpy` arrays is to import the `numpy` module using: `import numpy as np`. Now here are the few cool things you can do with `numpy` arrays:

(a) **Converting lists to `numpy` arrays:**
`arr = np.array([1,2,3,4,5])`

(b) **The `np.arange` function**
You can generate an array of a numerical sequence using
`arr1 = np.arange(1,6)`
This gives you an array of `[1 2 3 4 5]`
This function is similar to the Python `range()` function, except it actually returns an array-type object.

(c) **Doing maths with arrays**
You can also perform arithmetic on every single element in an array by writing say → `arr+1`
This will add 1 to every single element in the array yielding `[2 3 4 5 6]`
You can also do more complicated maths like → `np.sin(arr)**2`

WARNING: NEVER use for loops to do maths with arrays. (1) your code will be less efficient; (2) you will lose marks.

(d) **Array comprehension**
This is an elegant way of extracting (or checking if) elements in an array which satisfy a given condition.

- Let's say we would like to extract only elements that are greater than 2, we can write → `arr[arr>2]`
  This yields a new array with elements `[3 4 5]`
- We can also check if each element in the array satisfies a given condition, say we want to know which elements are greater than 2. We write → `arr>2`
  This yields `[False False True True True]`
- We can also extract elements in an array which satisfies the conditions of another array e.g. → Given that `y=arr**2`, we have `y=[1 4 9 16 25]`
  Then by writing `y[arr>2]`, we yield the array `[9 16 25]`.
  Essentially we have selected the elements in `y` whose indices correspond to those in `arr` that are greater than 2.

(e) **n-dimensional array axes** An array can, of course, be more complex than just one dimension. Axes are defined for arrays with more than one dimension. For example, a 2-dimensional array like:

```
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])
```

has two corresponding axes: the first running vertically downwards across rows (axis 0), and the second running horizontally across columns (axis 1).

(f) **Array of zeros** The `np.zeros` function return a new array of given shape and type, filled with zeros.
You feed in an argument which tells the function the shape of the new array
This argument can be either an integer (for a 1-D array) or a tuple (for an n-dimensional array).

Need something that is not written here? Hot tip: GOOGLE!

4.10 Errors and Exceptions (`try` and `except` statements)

Dealing with errors are inevitable when writing a program. In the first half of this section we will introduce you to a few common types of error that you may encounter.

Usually python will stop and generate an error message when it encounters an error. In the second part, we will show you how to handle errors with exceptions (i.e. without killing the program).

4.10.1 Common Error Types

Types of errors and the given examples are not limited to the following. There are many other possible errors but the ones listed here are the most common ones that you may encounter in this lab:

(a) **Syntax error**
When a certain statement is not in accordance with the prescribed usage e.g. missing brackets, missing a "colon" after if statements, misspelling statements like "eles" instead of "else etc.

(b) **Attribute error**
Raised when an attribute reference or assignment fails e.g. when `x=10` and you tried to perform `x.append(6)`

(c) **IndexError**
Raised when the index of a sequence is out of range e.g when you have a list `a=[0, 1, 2]` and you tried to perform `a[5]` where index 5 doesn't exist

(d) **NameError**
Raised when a variable is not found in the local or global scope.

(e) **TypeError**
Raised when a function or operation is applied to an object of an incorrect type e.g. when you try to divide an integer with a string

(f) **ValueError**
Raised when a function gets an argument of correct type but improper value e.g. `math.sqrt(-10)`

(g) **ZeroDivisionError**
Self explanatory

4.10.2 Try Except statements

Sometimes, there are errors that we expect and we would like the program to look past it and continue instead of terminating. This is were we employ exception handling using `try` and `except` statements.

We will use an example to show you how this works:

Code start

```
list1 = [[0, 1, 2], [4, 3, 7, 5], [2, 1, 0, 8], [3, 2, 1]]

for sublist in list1:
    try:
        print(sublist[3])
    except IndexError:
        continue

Program Output:
>>> 5
>>> 8
```

Code end

Basically the program aims to print the 3rd index for each of the sub-lists in `list1`. But we know that two of four of the sub-lists do not have a 3rd index (recalling zero is the first index). So here's a verbose breakdown of what we did in the example above:

- The `try` block lets you test a block of code for error
- Since we know the error that will be raised here is the `IndexError`, we want the program to make an exception for when this occurs
- The `except` block lets you handle the error
- The `except` block is only executed if the `IndexError` is raised, i.e. if any other error types are raised, the program will crash and proceed to raise whatever that other error is
- The `continue` statement basically tells the program to terminate that iteration of the `for`-loop and start the next iteration, essentially ignoring the sub-lists without a 3rd index

If you want your program to handle another type of error in a separate manner, you can create a second `except` block using an `except` statement on the type of error you want to handle e.g. `except TypeError:`

If you want your program to handle all types of error the same way, you can just use the `except` statement on its own i.e.

Code start

```
try:
    <code to be tested>
except:
    <error handling statement>
```

Code end

BUT BE VERY CAREFUL when you do this, you can potentially overlook a logical error or bug in your program!

4.11 Visualisation and Plotting: matplotlib

This section is an introductory guide on how to use `matplotlib` to visualise (i.e. plot) your results/data. The main sub-library you will be using is `pyplot`, which is a collection of functions that make `matplotlib` work like MATLAB. We typically import it as follows:

Code start

```
import matplotlib.pyplot as plt
```

Code end

Now, here are some basic functions that you will most definitely be using:

(a) **Creating a new figure**
`fig, ax = plt.figure()`
Fig refers to the entire figure, while ax is the axes.
You can also adjust the figure size by assigning a tuple containing (width, height) in inches to the parameter variable `figsize` e.g.:
`plt.figure(figsize=(10,8))`

(b) **Line plots**
`plt.plot(<xdata>, <ydata>)`
The x and y data are typically 1-D lists (or arrays) and they must have the same length (or shape)

(c) **Scatter plots**

`plt.scatter(<xdata>, <ydata>)`

(d) **Histograms**

`plt.hist(x, bins)`

x: Input values, this takes either a single array or a sequence of arrays which are not required to be of the same length

bins: If an integer is given, it defines the number of equal-width bins in the range. If a sequence (i.e. a list/array) is given, it defines the bin edges, including the left edge of the first bin and the right edge of the last bin; in this case, bins may be unequally spaced.

(e) **Axes labelling**

```
plt.xlabel('xlabel')
plt.ylabel('ylabel')
```

Demonstrator note: Always always ALWAYS label your axes. Otherwise the reader will have no clue what you're plotting. You will also lose marks for poor result presentation.

(f) **Legends**

You can plot and label multiple datasets in the same figure. Here is an sample code of how you can do this: `plt.plot(x1, y1, label='Plot 1')`

```
plt.plot(x2, y2, label='Plot 2')
plt.legend()
```

(g) **Displaying your plots (Interactive Window)**

`plt.show()`

Each `plt.figure()` will be shown in their respective windows.

WARNING: This command should only be used ONCE per python session and its usually at the very end of the script. Multiple `show()` commands can lead to unpredictable backend-dependent behaviors.

(h) **Saving your plots**

plt.savefig(filename, bbox_inches='tight')

Save the figure as filename, you can choose to save as a .pdf, .jpg, etc.

As mentioned before, everything listed here are really just the basics to using `matplotlib`, including the arguments to the functions we have introduced here - there are many more things you can do to customise your plots. For an extensive tutorial, you can refer to `https://matplotlib.org/stable/tutorials/index.html` for starters, and of course for other things GOOGLE is your friend.

4.12 SciPy and other useful packages

This section is pretty much just to inform you that the SciPy package exists along with various other useful libraries. The only SciPy package you will probably be `scipy.integrate` sub-package. But SciPy and many other Python libraries (Pandas, math etc.) consists of a large range of sub-libraries that can be extremely useful in scientific computation - so we suggest if you ever need a function that performs a relatively generic task when writing your scientific code, google it first before writing it yourself (this will save you A LOT of time which you can use to do much more useful things than recreating something that already exists).

5 General Programming Tips

5.1 Code structure and some pointers

For all labs, you will be provided with a skeleton code which give you a basic layout of the program that you will be writing, with some sections missing. Your job is to understand what the program does and to fill in these missing sections and add other functions. Basically, you do not need to worry how to structure your code from scratch but there are a few things we'd like you to take note of:

- Only ever run functions from the files containing `main()` functions. You should never need to create instances of functions or classes in other files. All functions are either directly imported and called in the main script OR used elsewhere in functions which will then be called in the main script.
- The functions with "NotImplementedError" are the ones you will need to edit for yourself. Most functions also have brief documentations that tells you what the function does. However, in some cases you may need to create new functions yourself to ensure that the code is fully functioning - so please do not fully rely on the skeleton code. Make your own judgements where appropriate and consult your demonstrators if you feel like you're overdoing things.
- **Use variables that are self explanatory** - try to avoid generic variables like "x" or "y" as it will be hard for the reader to keep up and make sense of your code.
- On a similar note, name your functions appropriately - don't call it things like "equation1()" etc.
- Optimise your code where you can within reason. This includes avoiding introducing unnecessary loops when numpy arrays can do the job much faster.

- **Comment your code!** Write your comments as you go, this is good for both you and the reader - don't leave this until the very end as you may forget what you did when you read back retrospectively. You also don't need to comment too excessively (i.e. on every line and every for/while loop etc.), commenting on blocks of code telling the user what it does will suffice, especially if you have clear variable names and function structure.

- **Clean up your code before submission** - Remove 'junk' code/debugging code that your program doesn't use. You should, however, leave in code that is needed to recreate the results you are analysing in your lab report. Make it easy to comment these out to only run one thing at a time.

5.2 Googling and Debugging

The ultimate skill that will get you through programming is knowing how and what to Google. As much as this sounds extremely trivial, knowing what to search will help you find reliable resources and solutions to your problems.

But to know what to search you will need to first understand your problem i.e. what errors/bugs are you facing. Read the terminal error message and make sure it's not just a silly syntax error or a typo. It literally tells you the chain of functions it went through that led to an error and what line of code it happened in. You can then either copy and paste the generic part of your error message into a search engine, or you can search something like "What does xxxError mean?" if you have no clue what's going on - usually quora or StackExchange (or Q&A platforms) will have the answer you're looking for, but of course sometimes more specific/complicated problems will require further digging. Nevertheless, your demonstrators are here to help, but your first reaction should be googling and chatting with your lab partners!

EXERCISE 3: Debug the seven programs in the 'debugging_exercises' folder that you unpacked earlier (you may have renamed this folder to something else). Each of the programs in the folder has a fatal error that will cause the program to terminate. On the very top of each file, there is a comment telling you what your goal is upon fixing the bugs.

6 Cheat sheet

Head to this page for a useful cheat sheet on python!

7 A micro lab on integration

Now, we're going to get hands on by doing a 'lab' from start to finish. You are expected to complete these steps during week 1 of semester to gain an idea of how to approach labs in general, how to set up a lab report and how to tackle problems you may encounter in computational labs. By completing and submitting a mini-report, you will be able to obtain feedback on how to do your best in the marked lab reports.

Each lab will give you background knowledge to get you familiar with the ideas and terminology that will be used.

This micro lab will get you to compare different methods of numerical integration, comparing the performance of basic coding with numpy arrays and pre-defined software packages.

Throughout this microlab, you will see Demonstrator's notes in red, these exist only in this lab to give you hints on how to do your best in the lab report. The green exercise sections are also added as extra help in case this is your first time coding.

7.1 Theory behind integration

In mathematics, an integral assigns numbers to functions in a way that describes displacement, area, volume, and other concepts that arise by combining infinitesimal data. The process of finding integrals is called integration. Along with differentiation, integration is a fundamental, essential operation of calculus, and serves as a tool to solve problems in mathematics and physics involving the area of an arbitrary shape, the length of a curve, and the volume of a solid, among others.

The integrals enumerated here are those termed definite integrals, which can be interpreted as the signed area of the region in the plane that is bounded by the graph of a given function between two points in the real line. Conventionally, areas above the horizontal axis of the plane are positive while areas below are negative. Integrals also refer to the concept of an antiderivative, a function whose derivative is the given function. In this case, they are called indefinite integrals. The fundamental theorem of calculus relates definite integrals with differentiation and provides a method to compute the definite integral of a function when its antiderivative is known.

In general, the integral of a real-valued function f(x) with respect to a real variable x on an interval [a, b] is written as

$$\int_a^b f(x)dx \tag{1}$$

The integral sign $\int$ represents integration. The symbol dx, called the differential of the variable x, indicates that the variable of integration is x. The function f(x) is called the integrand, the points a and b are called the limits (or bounds) of integration, and the integral is said to be over the interval $[a, b]$, called the interval of integration. A function is said to be integrable if its integral over its domain is finite. If limits are specified, the integral is called a definite integral.

The integral of the function f can be calculated analytically by determining its antiderivate, F:

$$\int_a^b f(x)dx = F(b) - F(a) \tag{2}$$

Demonstrator's note: In the introduction of your report, you'll want to summarise this in your own words. You might also not be able to fully flesh out an aim until you have read through everything, which you should always do before starting the lab.

QUESTION 1: Analytically calculate $\int_0^1 3x^2 + 4x^3 - 2\,dx$
Demonstrator's note: This can be done in your introduction or theory section

7.2 Numerical integration via rectangles

If the antiderivative is complicated, there are several methods to approximate the area under the curve numerically. One method is the rectangle method. The rectangle method relies on dividing the region under the function into a series of rectangles corresponding to function values and multiplies by the step width to find the sum. The more subdivisions you use, the more accurate the estimation will be.
Demonstrator's note: You would want to include a diagram of how this works in your theory or method section

Exercise 4:

1. Create and test a function that returns the output of the integrand in Question 1
2. Define a rectangular method function that takes the following arguments: an integrand f, a lower bound a, an upper bound b, and a number of sub-divisions n
3. Inside this function, define a 'width' using the bounds and number of sub-divisions
4. Create a loop over each sub-division, defining a value of x and $f(x)$
5. Use the for-loop, each $f(x)$ value and the width to create a sum of each sub-division area
6. Return the total area

Demonstrator's note: When writing your code you should include useful names, comments and function descriptions. For example, the integration function should look something like this:

Code start

```
def rectangle_integrate(f, a, b, n):
    """
    A function for integrating the function f via the rectangle
    method, between the bounds a and b with n sub-divisions,
    using for-loops without numpy.
    """
    width = ...

    # Loop through each sub-division and add up all the areas
    total_area = 0
    for ...:
        x = ...
        length = f(x)
        total_area += length*width
    return total_area
```

Code end

QUESTION 2: How long does it take to numerically calculate the integral from Question 1 using the rectangle method with 10^3 subdivisions, without using the numpy package? How long for it to be correct to roughly 5 decimal places?
Hint: Define the error as the absolute value of the algorithm's output, minus the analytic solution. The error should be $< 10^{-5}$
Demonstrator's note: You would be expected to chat with your partners or google something like 'How long does something take in python' to find code examples that you can test. Also, make sure you have a main() function where you can run all other functions

EXERCISE 5:

1. Create another rectangular method function that will take the same arguments as the one from the previous exercise
2. Define an array of x values using the numpy linspace function
3. Define a width based on the difference between two neighbouring x values
4. Define an array for $f(x)$
5. Define an array for the areas
6. Return the total area

QUESTION 3: How long does it take for the integration to be accurate to roughly 5 decimal places using numpy?

Your task here is to determine how much time is taken for each of these algorithms to complete as the number of calculations increases, and how accurate they are.

Exercise 6: This exercise will help with understanding how to best utilise a combination of for loops and numpy arrays

1. Create an array of the number of subdivisions that will be tested, use the numpy logspace function, keeping the maximum below $\approx 10^9$
2. Create a loop that keeps track of the rectangular integration function output for each number of subdivisions along with keeping track of how long it takes to finish
   HINT: Google is your friend for any errors that might show up with n...
3. Define an array that gives the error of each of the outputs (See question 2 for the definition of error)
4. Create a scatter plot of the error against the number of sub-divisions, using a log-log scale
5. Use the numpy polyfit function to obtain a line of best fit for the log values
6. Plot the line of best fit on the scatter plot
   HINT: You need to plot the non-log values, so you should do some maths to get the correct expression
   Demonstrator's note: You should make sure to write down the formula you get in your log books along with the maths.
7. Repeat steps 4-6 for the error against time taken
8. Repeat steps 2-7 for the other rectangle method algorithm

Question 4: How many subdivisions would you need to achieve float accuracy and how long would it take?
Demonstrator's note: If you don't know what float accuracy means, chat with the people around you, or try to google it. If you are still unsure, ask the demonstrator.

7.3 No need to reinvent the wheel

Luckily, we no longer have to calculate integrations through the rectangle method. Another method is Simpson's rule:

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\Big[f(a) + 4f(\frac{a+b}{2}) + f(b)\Big] \tag{3}$$

Demonstrator's note: It would be a good idea to discuss how this rule comes about.
And finally, there is scipy's quad integration.

Question 5: How do these two methods compare in computation time and accuracy compared to the previous methods? What if you used a more complicated function for f involving fractional powers?
Demonstrator's note: This would be a good time to make sure your code is structured neatly so that it is easy to swap out functions and pick which sections to run.

Question 6: Using the knowledge that $\int_0^1 4\sqrt{1-x^2} = \pi$, extrapolate how long it would take using the rectangular method algorithms to calculate π to the same accuracy as scipy's quad integration.

Demonstrator's note: Make sure to write a conclusion that discusses the learning goals here with important results and insights you have found