# Introduction to Python

Python is a powerful high-level interpreted language. In this lab session, you are expected to familiarise yourself with Python and Jupyter Environment.

# Installation

The easiest way for a beginner to get started with Jupyter Notebooks is by installing Anaconda, which comes with the most widely used Python distribution for data science and comes pre-loaded with all the most popular libraries and tools. Anaconda includes Numpy, Pandas, and Matplotlib, plus 1000 more libraries.

You can download Anaconda directly from: https://www.anaconda.com/download. Install Anaconda by following the instructions on the download page.

If you are a more experienced user and having already installed the Python, you may prefer to use pip package to manage your packages manually: pip3 install jupyter

# What is an ipynb file?

It is an editable project file, the so-called **Notebook** file. Each time you create a new file, a new **.ipynb** file will be created.

More technically, an ipynb file is a text file that describes the contents of the notebook file using JSON format. Each cell and its associated contents (including images) will be transformed into strings of text along with some metadata.

# Notebook user interface

What you see is a notebook interface in front of you. Check out the menus to get a feel for it. Kernel and Cell terms are key to understand Jupyter. A **kernel** is a **computational engine** that executes the code in the notebook. A **cell** is a container for text to be displayed or code to be executed by the notebook's kernel depending on the mode chosen (see the next section for *Code* or *Markdown* cells).

# Let's have a look at the Cells

- A *code cell* contains the code to be run, where the output is displayed below the code cell.
- A *markdown cell* contains formatted text and displays the output below the markdown cell.

Let us now test it out with a hello world message. Type print('Hello World') into a cell you create below and click the run button. Finally, repeat the same process by simply pressing Ctrl+Enter while you are at the cell to print "Hello World".

Did you realise that a [ ] number is given after you run the code? While code cells have this numerical label, markdown cells do not. Now, keep running the code in the same cell repeatedly, and observe the label. Did you notice that the label continues increasing. This label is important in consecutive execution of the codes in different cells to diagnose if a particular cell (with current [ ] label number) creates an issue. This will be clearer in Kernel Section.

# Keyboard options

Go ahead and try the following keyboard options to familiarise yourself. You do not need to memorise them all but some very useful shortcuts are as follows.

- You can toggle between edit and command mode by ESC and Enter.
- When you are in command mode:
  - Up and Down keys help you move between cells.
  - Press A and B to create a cell above or below.
  - Pressing M will transform the active cell to markdown cell.
  - Pressing Y will transform the active cell to code cell.
  - Pressing CTRL + / will comment out in code cell.
  - Pressing D consecutively (two times) will delete the active cell.
  - Pressing Z will undo the removed cell.
  - Holding Shift button and pressing Up and Down will help you select multiple cells and additionally pressing Shift+M will merge all the cells selected.

Now, create a markdown cell and move the next section.

# Markdown

Markdown is a lightweight, easy to learn markup language for formatting text. It's syntax is similar to HTML. Remember this notebook is in part created by Markdown cells and thus you can observe some of the usage of the followings. Go ahead and make changes to the following text by looking at the editable markdown mode (you can double click or press Enter while on the cell).

# This is a level 1 heading

## This is a level 2 heading

This is some plain text that forms a paragraph. Add emphasis via **bold** and **bold**, or *italic* and *italic*.

Paragraphs must be separated by an empty line.

- Sometimes we want to include lists.
- Which can be bulleted using asterisks.

1. Lists can also be numbered.
2. If we want an ordered list.

It is possible to include hyperlinks

# Kernel

A kernel is run behind every notebook. Notebook pertains the kernel to the whole document. What does that mean? For example, you can import a library in one cell and use in the next cells as long as the first code cell is run for the libray to be imported. See the below cells.

In [ ]:
```
# We will come to functions later. However, this is to show how kernels and
# Define a function named square, which returns x^2.
import numpy as np
def square(x):
    return x * x
```

In [ ]:
```
# Observe that we can call the random and square functions from the cell pre
# If you run this code consecutively, in each run the result will change dep
x = np.random.randint(1, 10)
y = square(x)
print('%d squared is %d' % (x, y))
```

In [ ]:
```
# Observe that if you solely run the print function, it uses the latest valu
# Run the print several times and you will see that the result does not char
print('%d squared is %d' % (x, y))
```

In [ ]:
```
# Now, we manipulate the value of y to be 3.
# What do you think it will print out?
# Hint: you will need to find the value of x that was from the latest kernel
y=3
print('%d squared is %d' % (x, y))
```

# Notes

- Most of the time when you create a notebook, the flow will be top-to-bottom. You can always go back to the concerned [ ] label number to fix the problems if any issues occur at the time of the kernel run. You can also use the reset options as follows:
  - Restart: Restarts the kernel, clearing all the variables that were defined.
  - Restart & Clear Output: In addition to Restart, it will also clear the output displayed below your code cells.
  - Restart & Run All: In addition to Restart, it will run all your cells in order from first to the last cell.
  - Interrupt: If your kernel is ever stuck during a computation, you can choose the Interrupt option to stop it.

# Variables

Integers

In [ ]:
```
a = 1
b = 3
print("Sum, difference, division:", a + b, a - b, a // b)
```

Floting point numbers

In [ ]:
```
print("Floating point division:", 1.0 / 2.0)
```

Complex numbers

In [ ]:
```
print("Complex numbers:", 1.0 + 1.0j)
```

Booleans

In [ ]:
```
a = True
b = False
print("Boolean operations:", a or b, a and b, not a)
```

Strings

In [ ]:
```
s = "n"
print("String:", s)
```

Single quotes can also be used

In [ ]:
```
a = 'This is a string too'
```

We can split long strings like this

In [ ]:
```
a = ("Very very very "
     "long long long "
     "string in Python"
    )
a
```

Or use multiline string

In [ ]:
```
a = """this
is
multiline
string"""
a
```

String concatenation

In [ ]:
```
"str" + "ing"
```

Some of useful string methods:

In [ ]:
```
a = ""
dir(a)
```

Try following methods: `.endswith`, `.join`, `.capitalize`

String formatting

In [ ]:
```
"This is a number {}, this is another number {}!".format(10, 20)
```

You can specify how number is formatted

In [ ]:
```
"This is pi {:0.2f}!".format(3.1415)
```

Format strings look like this

In [ ]:
```
f"This is sum of 2 and 3: {2 + 3}"
```

# Simple data structures: lists, maps, sets, tuples

Lists are designed to store a number of ordered values.

## List

In [ ]:
```
array = [1, 4, 2, 3, 8, 7, 6, 5]
array
```

Addressing list by index

In [ ]:
```
array[0]
```

Slice is a sub-sequence of a list

In [ ]:
```
array[1:5]
```

End-less slices take either prefix

In [ ]:
```
array[:5]
```

or suffix

In [ ]:
```
array[5:]
```

Third argument to the slice is the step size

In [ ]:
```
[1, 4, 2, 3, 8, 7, 6, 5]
array[2:7:2]
```

Lists may contain values of different types

In [ ]:
```
len([1, 1e-8, "Hello", [9, 8]])
```

## Maps

Maps (dictionaries) can store relations between pairs of values

In [ ]:
```
m = {"height": 100.,
     "width": 20.,
     "depth": 10.}
m
dict(height=100.)
```

Retrieving value by key

In [ ]:
```
m["width"]
```

Checking that a map contains a key

In [ ]:
```
"name" in m
```

Add a new key-value pair

In [ ]:
```
m["name"] = "rectangle"
m
```

Or change existing value

In [ ]:
```
m["name"] = "RECTANGLE"
m
```

Remove key/value

In [ ]:
```
m.pop("name")
```

In [ ]:
```
m
```

## Tuples

Tuples are similar to lists but are immutable -- they cannot be altered.

In [ ]:
```
my_array = [1, 2, 3]
my_tuple = (1, 2, 3)

# This is OK
my_array[0] = 100

# This will raise an exception
my_tuple[0] = 100
```

In [ ]:
```
a = (1, 2)
b = (3, 4)
#(1, 2) + (3, 4)
a+b
```

## Sets

Sets are unordered collections that support fast search, insertion, deletion and union.

In [ ]:
```
animals = {"cat", "dog", "elephant"}
animals
```

Check that element is in set

In [ ]:
```
"cat" in animals
```

Perform set operations: union, intersection, etc

In [ ]:
```
animals.union({"zebra", "llama"})
```

# Control flow

Branching

In [ ]:
```
a = int(input())
if a > 6:
    print("a is greater than 6")
elif a < 3:
    print("a is less than 3")
else:
    print("a is between 3 and 6")
```

Loops

In [ ]:
```
for i, j in enumerate(["cat", "dog"]):
    print(i, j)
```

Useful functions for looping:

- `range`
- `enumerate`
- `zip`

Iterating a dictionary

In [ ]:
```
for k, v in m.items():
    print(k, v)
    if k == "width":
        continue
    # long processing
    print("again", v)
```

While loop

*It is very rare that you need to use while loop. Following example is very not pythonic!*

In [ ]:
```
stop = False
i = 10
while not stop:
    i += 1
    if i % 10 == 0:
        stop = True

print(i)
```

# List comprehensions

In [ ]:
```
[i + 1 for i in [1, 2, 3] if i != 2]
```

It works with dictionaries too

In [ ]:
```
{i: i + 1 for i in [1, 2, 3]}
```

# Functions

Defining functions

In [ ]:
```
def is_odd(a):
    return a % 2 == 0

is_odd(2)
```

Functions can be defined inside functions

In [ ]:
```
def is_odd(a):

    def is_divisible(number, base):
        return number % base == 0

    return is_divisible(a, 2)

is_odd(2)
is_divisible(6, 5)
```

You can provide default arguments.

In [ ]:
```
def add_or_subtract(first, second, operation="sum"):
    if operation == "sum":
        return first + second
    elif operation == "sub":
        return first - second
    else:
        print("Operation not permitted")

add_or_subtract(first=3, second=4, operation="sum")
```

Varargs: variable size arguments

In [ ]:
```
def sum_all(*args):
    # args is a list of arguments
    result = 0
    for arg in args:
        result += arg
    return result
```

```
# Call vararg function
print("Sum of all integers up to 10 =", sum_all(1, 2, 3, 4, 5, 6, 7, 8, 9))
```

Keyword arguments

In [ ]:
```
def print_pairs(**kwargs):
    # kwargs is a map
    for k, v in kwargs.items():
        print(k, v)

print_pairs(a=1, b=2)
```

Keyword only arguments

In [ ]:
```
def create_car(*, speed, size):
    print("Car created with speed", speed, "and size", size)

create_car(speed=9, size=3)
```

Functions as parameters

It is possible to pass a function as an argument, operation here is assumed to be a function

In [ ]:
```
def reduce(array, operation):
    result = 0
    for k, v in enumerate(array):
        if k == 0:
            result = v
        else:
            result = operation(v, result)

    return result
```

Apply the function with another function `add_or_subtract`

In [ ]:
```
one_to_nine = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(one_to_nine)

print("Sum of the array")
#The operation is infered from the default parameter of add_or_subtract
reduce(one_to_nine, add_or_subtract)
```

Lambdas

A function can also be defined anonymously

In [ ]:
```
print("Product of the array")
reduce(one_to_nine, lambda x, y : x * y)
```

```
prod(5, 6)
```

Closures

A function can return another function with specific behaviours depending on the arguments

In [ ]:
```
def get_loss(op_reduce, op_foreach):

    def loss(a, b):
        c = []
        for av, bv in zip(a, b):
            c.append(op_foreach(av, bv))
        return op_reduce(c)

    return loss
```

This function can help to define mean squared error

In [ ]:
```
mse_loss = get_loss(lambda x : sum(x) / len(x),
                    lambda a, b : (a - b) ** 2)
```

Or mean absolute error

In [ ]:
```
mae_loss = get_loss(lambda x : sum(x) / len(x),
                    lambda a, b : abs(a - b))
```

We can check that it works as intended

In [ ]:
```
list1 = [0, 1, 1, 3, 0, 2, 3]
list2 = [1, 1, 2, 0, 0, 2, 3]

list_mse = mse_loss(list1, list2)
list_mae = mae_loss(list1, list2)

print("Two lists:\n", list1, "\n", list2)
print("MSE Loss: {}\nMAE Loss: {}".format(list_mse, list_mae))
```

# Exceptions

Software applications don't always work flawlessly. Even with thorough troubleshooting and multiple testing phases, they can still malfunction. Problems like bad data, unstable network connections, corrupted databases, memory overloads, and unexpected user inputs can disrupt an application's normal operations. When an application can't function as expected due to these issues, it experiences what's called an exception. As a programmer, it's your responsibility to catch and manage these exceptions to ensure your application keeps running smoothly.

In [ ]:
```python
# Throw exception
from datetime import datetime

current_date = datetime.now()
print("Current date is: " + current_date.strftime('%Y-%m-%d'))

dateinput = input("Enter date in yyyy-mm-dd format: ")
# We are not checking for the date input format here
date_provided = datetime.strptime(dateinput, '%Y-%m-%d')
print("Date provided is: " + date_provided.strftime('%Y-%m-%d'))

# Lets raise an exception because we know this can be a potential issue
if date_provided.date() < current_date.date():
    raise Exception("Date provided can't be in the past")
```

In [ ]:
```python
# Catch exceptions
try:
    raise ValueError
except ValueError:
    print("Do something else")
finally:
    print("This part runs always. It is useful for closing files or "
          "releasing other resources")
```

In [ ]:
```python
# Catch exception example
import rollbar
num0 = 10

try:
    num1 = int(input("Enter 1st number:"))
    num2 = int(input("Enter 2nd number:"))
except ValueError as ve:
    print(ve)
    rollbar.report_exc_info()
    exit()
except ZeroDivisionError as zde:
    print(zde)
    rollbar.report_exc_info()
    exit()
except TypeError as te:
```

```
    print(te)
    rollbar.report_exc_info()
    exit()
except:
    print('Unexpected Error!')
    rollbar.report_exc_info()
    exit()
else:
    result = (num1 * num2)/(num0 * num2)
    print(result)
```

# Classes

Class definition

In [ ]:
```
class Shape:
    pass
```

`shape` is an object of class `Shape`

In [ ]:
```
shape = Shape()
type(shape)
```

In legacy python we used to write

`class Shape(object):`
This is not needed anymore unless you expect someone to run your code in legacy environment

More on class definitions

In [ ]:
```
class Shape:
    class_field = 9

    def __init__(self, name):
        self.name = name
        self.value = 42

    def method(self, a):
        return a * 2 + self.value

shape = Shape("name")
shape.method(7)
```

In [ ]:
```
# A Sample class with init method
class Person:

    # init method or constructor
    def __init__(self, name):
        self.name = name

    # Sample Method
```

```
    def say_hi(self):
        print('Hello, my name is', self.name)


p = Person('Halil')
p.say_hi()
```

In [ ]:
```
# A Sample class with init method
class Person:

    # init method or constructor
    def __init__(self, name):
        self.name = name

    # Sample Method
    def say_hi(self):
        print('Hello, my name is', self.name)


# Creating different objects
p1 = Person('Halil')
p2 = Person('Aboubaker')
p3 = Person('Serengul')

p1.say_hi()
p2.say_hi()
p3.say_hi()
```

# Imports

Adding new packages in python is very easy and many packages are available from the box. If you want some library, there is a good chance that someone else wrote it already.

Generally, import statement looks like

In [ ]:
```
import time

time.time()
```

You can specify what parts of the package you want to import

In [ ]:
```
from time import time, sleep

print(time())
sleep(2)
print(time())
```

# Other useful features

Multiple assignment

It works with any kind of list-like objects!

In [ ]:
```
a, b = [10, 11]
print(a+b)
```

Starred assignment expressions

In [ ]:
```
a, *b = [1, 2, 3, 4]
print(b)
print(a)
```

This works for prefixes and sufixes

In [ ]:
```
*a, b, c = [1, 2, 3, 4]
print(a)
```

# Exercises

## Exercise 1

Write a function that samples a uniform random number from `a` to `b`.

Use function `random.random` from package `random`. "a + (b-a) * random()" can be returned as a sample shown at the documentation page here Remember to use exceptions! Throw or Catch exception. Test your function by calling one_sample(start, end).

In [ ]:
```

```

## Exercise 2

Write a function that creates a list of length `n` of samples like in Excercise 1. In this example, we expect to see `n` number of samples. Remember to use exceptions! Run the exercise multiple times and observe how the result changes with the range. Test your function by calling multiple_samples(start, end, n).

In [ ]:
```

```

## Exercise 3

Write a function that computes an average of any list of numbers within a certain range. The array length will be passed to the function using `random.randint()` (length and array values range between 1 and 20). Name the function

`mean(length)` and inside the function define an initial array, namely `my_array=[]` and write the generated random values to this array given the lenght and print the mean. Remember length will be randomly defined before being passed to the function mean, but you can initially start working with a fixed lenght for simplicity. Run as many as you like to see the changes in the output.

In [ ]:

## Exercise 4

Import `Pandas` and create a simple `Pandas DataFrame`.

In [ ]:

## Exercise 5

Import `matplotlib` and `numpy` to create a linear line from `(1,2) to (9,12)` coordinates.

In [ ]:

## Exercise 6

Import `numpy` and create an array with non-repeting elements from 0 to 14 and reshape it as a 3x5 matrix. You can use `arange` and `reshape` functions.

In [ ]: