UCL MECHANICAL ENGINEERING

UCL ENGINEERING
Change the world

Mechanical Engineering And Practical Skills I (MECH0004)

Introduction to Python Programming Language

Dr Lama Hamadeh

2022 - 2023

Contents

1 Introduction

Object-Oriented programming (OOP) is a widely used concept to write powerful applications. Oriented simply means *directed toward*. So object-oriented functionally directed towards modeling objects. This is one of the many techniques used for modeling complex systems by describing a collection of interacting objects via their data and behaviour [1]. It has some advantages over other design patterns. Development is faster and cheaper, with better software maintainability. This, in turn, leads to higher-quality software, which is also extensible with new methods and attributes. OOP is a programming paradigm that provides a means of structuring programs so that properties and behaviors are bundled into individual objects. It uses the concept of objects and classes. A class can be thought of as a 'blueprint' for objects. An object has two characteristics: attributes (characteristics they possess) and behavior(actions they perform). An example of a class is the class Dog. Don't think of it as a specific dog, or your own dog. We're describing what a dog *is* and can *do*, in general. Dogs usually have a name and age; these are instance attributes. Dogs can also bark; this is a method.

A very important example of a programming language that supports OOP is Python. Python is a general-purpose, versatile, and powerful programming language. It's concise and easy to read. Python offers a number of benefits compared to other programming languages like Java, C++ or R. It's a dynamic language, with high-level data types. This means that development happens much faster than with Java or C++. It does not require the programmer to declare types of variables and arguments. This also makes Python easier to understand and learn for beginners, its code being more readable and intuitive.

These workshops are designed to provide an introductory knowledge to the basics of Python programming language. You will start by doing basic arithmetic and use variables in Python. You will learn how to handle data structures such as Python lists and Numpy arrays. You will also learn about loops, conditions and functions. To top it off, you will learn about visualizing your data using Python.

1.1 Why Python?

Python is a general-purpose and in-demand programming language that has become one of the most popular languages in the world. In recent years, as shown in Fig(1), Python has demonstrated a rapid growth as the introductory language in computer science courses and is becoming a preferable programming language in industry and academia [2], [3].

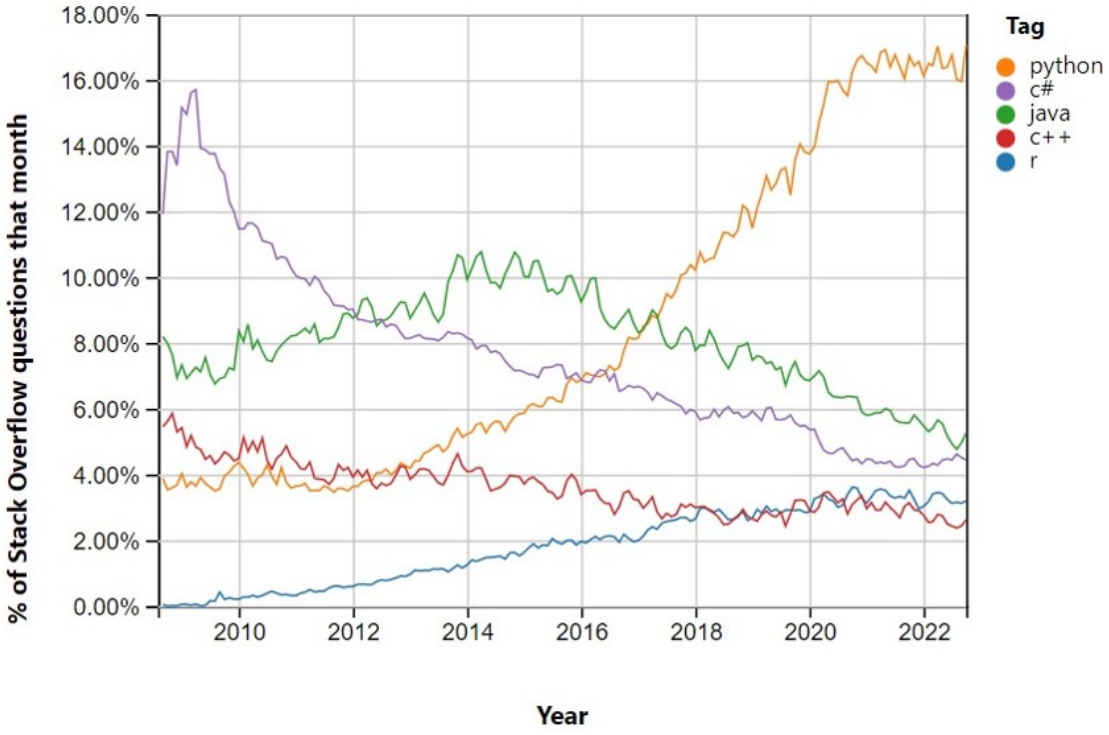


Figure 1: A snapshot of how programming languages have trended over time based on their tags on Stack Overflow since 2008.

Python has libraries for data loading, visualisation, statistics, natural language processing, image processing, and more. This vast ecosystem of tools, packages, and libraries addresses a wide-ranging number of programming scenarios and provides scientists and engineers with a large array of general- and special-purpose functionality. In addition, Python has an easy-to-learn structure, runs on multiple operating systems, license-fees free, and has an excellent online documentation. Moreover, Python programs tend to be much shorter than equivalent programs in C or C++ or Java.This is because Python is a high-level language. It has data types that allow you to express complex operations in a very concise manner. This surely makes Python stands out from all the other programming languages.

1.2 Python 2 Vs Python 3

There are two major versions of Python that are widely used at the moment: Python 2 (more precisely, 2.7) and Python 3 (with the latest release being 3.9 at the time of writing). This sometimes leads to some confusion of which version to use or learn. However, when looking at the current state of both versions, the

answer is relatively easy. Python 2 is outdated and no longer actively developed, whereas Python 3 contains major changes and is constantly in-demand [4]. While Python 2 is still in use for legacy reasons, Python 3 is the current standard. It is very worth mentioning that these two versions are not compatible. This means that if you write your code in Python 3, somebody running Python 2 will not be able to run that code.

In these workshops, we will be using Python 3 and that due to the fact that all new standard library improvements are only available by default in Python 3. Python 3 is also easier for newcomers to learn, and several aspects of the core language are more consistent than those in Python 2.

1.3 Python Essential Libraries

Python is an ocean of libraries that serve various purposes. This means there's a good chance that whatever you're trying to build, there's already a package (or a library) that can make the development easier for you. Normally, as we shall see later, Python requires you to import this library into your project first before you start coding. Importing a library is like getting a piece of lab equipment out of a storage locker and setting it up on the bench for use in a project [4]. As a Python beginner, you should familiarise yourself with the most important, useful and ubiquitous Python packages since they will come up more often. In the following, we will outline the main and most occurring and used libraries in Python that you should know [5].

1.3.1 Data Processing and Modeling

1.3.1.1 numpy It is one of the fundamental packages for scientific computing in Python. **numpy** provides tools to help build multi-dimensional arrays and perform calculations on the data stored in them. You can solve algebraic formulae, perform common statistical operations, and much more.

1.3.1.2 scipy As the name suggests, **scipy** is mainly used for its scientific functions and mathematical functions derived from **numpy**. **scipy** contains modules for optimization, linear algebra, integration, interpolation, special functions, Fast Fourier Transform (FFT), signal and image processing, ordinary differential equations (ODE) solvers and other scientific computing tasks common in science and engineering.

1.3.1.3 pandas A Major part of an engineer's job is to understand and clean data, i.e., data exploration and manipulation. **pandas** is primarily used for data analysis, and it is one of the most commonly used Python libraries. It provides

you with some of the most useful set of tools to explore, clean, and analyse your data. **pandas** is a library created to help developers work with "labeled" and "relational" data intuitively. It's based on two main data structures: "Series" (one-dimensional, like a list of items) and "DataFrames" (two-dimensional, like a table with multiple columns). **pandas** allows converting data structures to DataFrame objects, handling missing data, and adding/deleting columns from DataFrame, imputing missing files, and plotting data.

1.3.2 Data Visualisation

1.3.2.1 matplotlib This is the primary scientific plotting library in Python. It provides functions to generate data visualisations such as line charts, histograms, scatter plots, and so on. Data visualization gives us a clear idea of what the information means by giving it visual context through maps graphs. This makes it easier to identify trends, patterns, and outliers within large data sets.

2 Anaconda Distribution & Jupiter Notebook

As mentioned before, Python has a sea of libraries to perform and tackle several scientific computing scenarios. But instead of installing hundreds of packages manually one at a time, we will be using a Python distribution. A distribution consists of the core Python packages and several hundred modules, all working seamlessly together. All of this is available through a single download. There are currently several Python integrated development environments (IDEs) that allows one to write, test, and debug Python software, e.g., IDLE, Microsoft Visual Studio, PyCharm, PyDev, Thonny, etc. The one that we will be using in these workshops is called Anaconda distribution. One of Anaconda's very useful development environments is Jupyter Notebook. Jupyter Notebook is a free, open-source, interactive web tool known as a computational notebook, which researchers can use to combine software code, computational output, explanatory text and multimedia resources in a single document. Computational notebooks have been around for decades, but Jupyter in particular has exploded in popularity over the past couple of years due to its flexibility and easy-to-use interface. You can use Jupyter Notebooks for nearly all sorts of science and engineering tasks including data cleaning and transformation, numerical simulation, exploratory data analysis, data visualization, statistical modeling, machine learning, deep learning, and much more [4].

2.1 Download Anaconda

Anaconda can be easily downloaded onto your computer/laptop using UCL software database following the next steps, as shown in Fig(2):

1. Open UCL Software Database using this link: `https://swdb.ucl.ac.uk/`, and then enter your UCL username and password.
2. In the "Search" box, type "Anaconda" and then hit "enter" to look for the package in the dataset.
3. The search should bring back one result: "Anaconda Python". Click on it to view its details.
4. The main page of the package details has several horizontal tabs that provide different tools, e.g., general, licences, availability, etc. Scroll down and click on "Downloads".
5. The "Downloads" tab has a link that directs the user to Anaconda website.
6. Locate your download and install Anaconda.

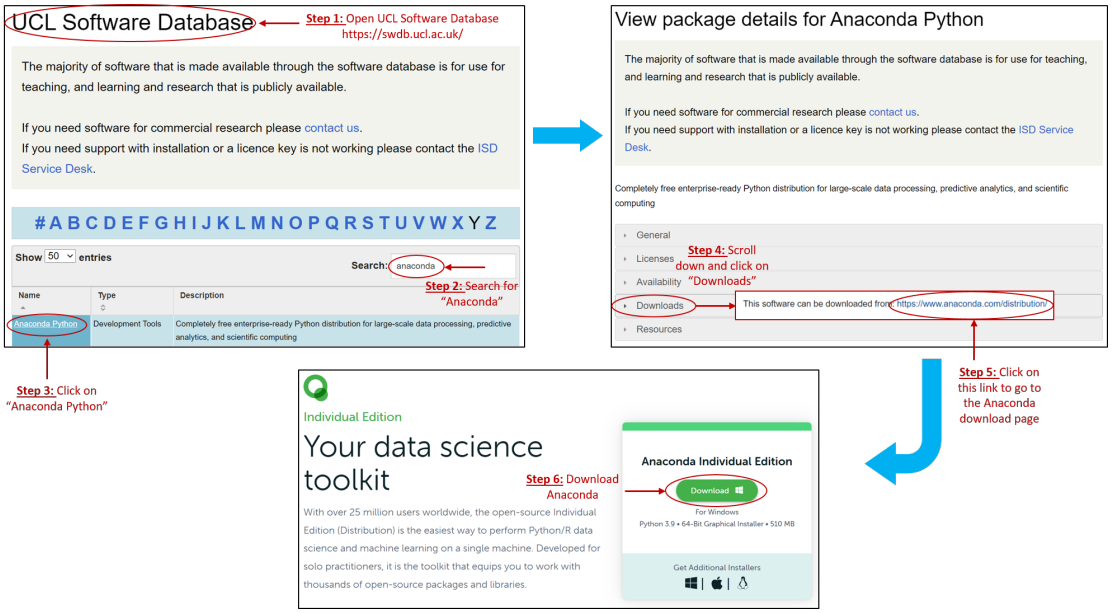


Figure 2: Download Anaconda Distribution.

2.2 Launch Jupyter Notebook

You can launch Jupyter Notebook easily in a straightforward manner by double-clicking on Anaconda icon and then scroll down to find Jupyter Notebook application, and then hit "Launch" to open it.

Once you are in the Jupyter Notebook interface, you can see all the files in your current directory. All Jupyter Notebooks are identifiable by the notebook icon next to their name. If you already have a Jupyter Notebook in your current directory that you want to view, find it in your files list and click it to open. Notebooks currently running will have a green icon, while non-running ones will be grey. To find all currently running notebooks, click on the Running tab to see a list.

2.3 Create, Understand and Run a Jupyter Notebook

To create a new notebook, go to "New" and select Notebook - Python 3. If you have other Jupyter Notebooks on your system that you want to use, you can click "Upload" and navigate to that particular file.

When you open a new Jupyter notebook, you will notice that it contains a rectangular cell. Cells are how notebooks are structured and are the areas where you write your code. The browser then passes the code to a back-end "kernel"

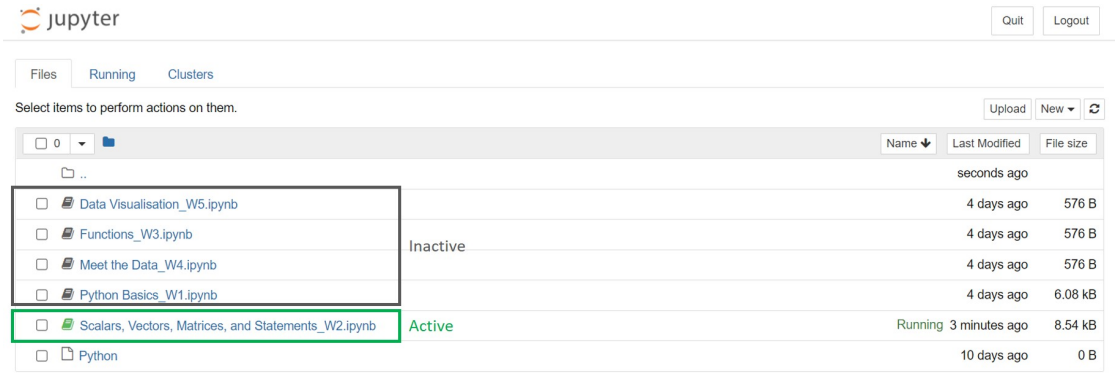


Figure 3: Jupyter Notebook Interface

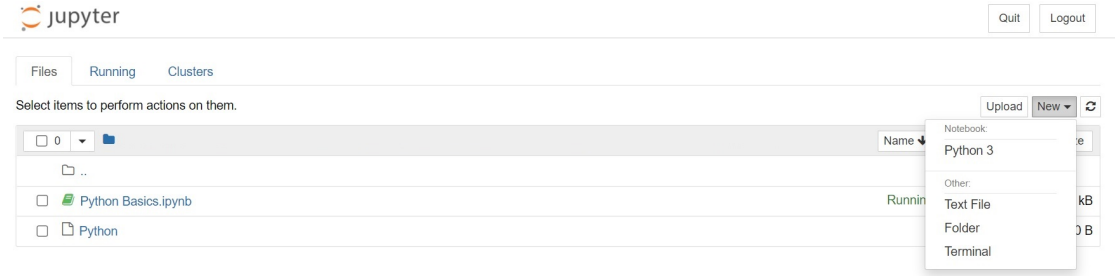


Figure 4: Create a new Jupyter Notebook file

which runs the code and returns the results. This rectangular-cells structure has a major flexibility when it comes to managing your code. Rather than writing and re-writing an entire program, you can write lines of code and run them one at a time. Then, if you need to make a change, you can go back and make your edit and rerun the program again, all in the same window. To run a piece of code, click on the cell to select it, then press `SHIFT+ENTER`. After your run a cell, the output of the cell's code will appear in the space below. The toolbar has several shortcut buttons for popular actions. From left to right: save, add a new cell, cut selected cells, copy selected cells, paste cells below, move selected cells up, move selected cells down, run, interrupt the kernel, restart the kernel, a dropdown that allows you to change the cell type, and a shortcut to open the command palette.

The navigation bar also contains several tabs with multiple options. From left to right: File, Edit, View, Insert, Cell, Kernel, Widgets, Help.

To create new cells, use the plus (+) button in the toolbar or from the "Cell" tab in the navigation bar. To cut, copy, delete or just generally edit cells - select

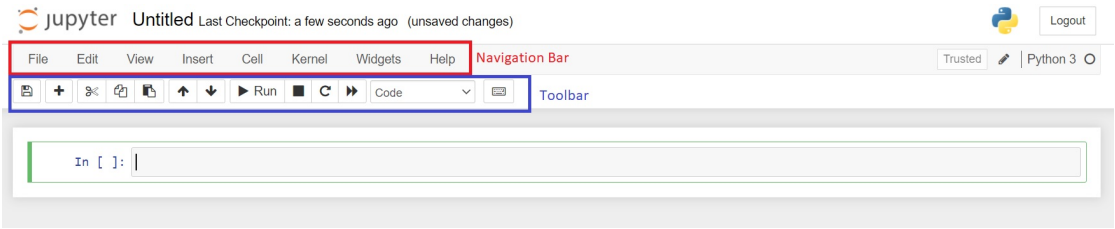


Figure 5: Jupyter Notebook Toolbar and Navigation Bar

the cell you want to modify and go to the "Edit" tab in the navigation bar to see your options.

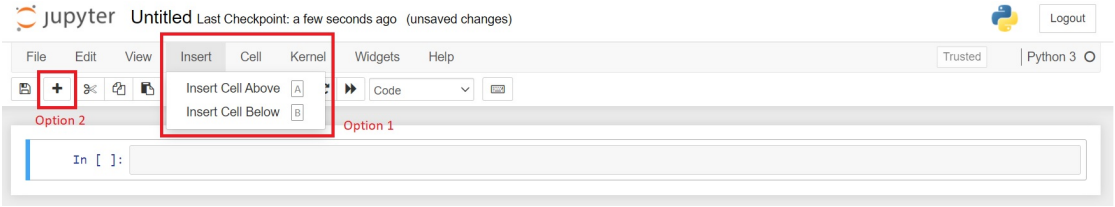


Figure 6: Two options on how to create a new cell.

In addition to running lines of code, you can also include text-only cells that use Markdown format to organize your notebooks. Cells in Jupyter are set up to be in code mode by default. To change its type to a Markdown, there are two options to follow: either choosing the "Markdown" options from the toolbar, or go the "Cell" tab in the navigation bar, choose "Cell Type" and then "Markdown" as its type.

Jupyter Notebook files are saved as you go. They will exist in your directory as a JSON file with the extension `.ipynb`. You can also export Jupyter Notebooks in other formats, such as HTML. To do so, go to the "File" menu, scroll down to "Download as" and select the type of file you are looking for as shown in Fig(8).

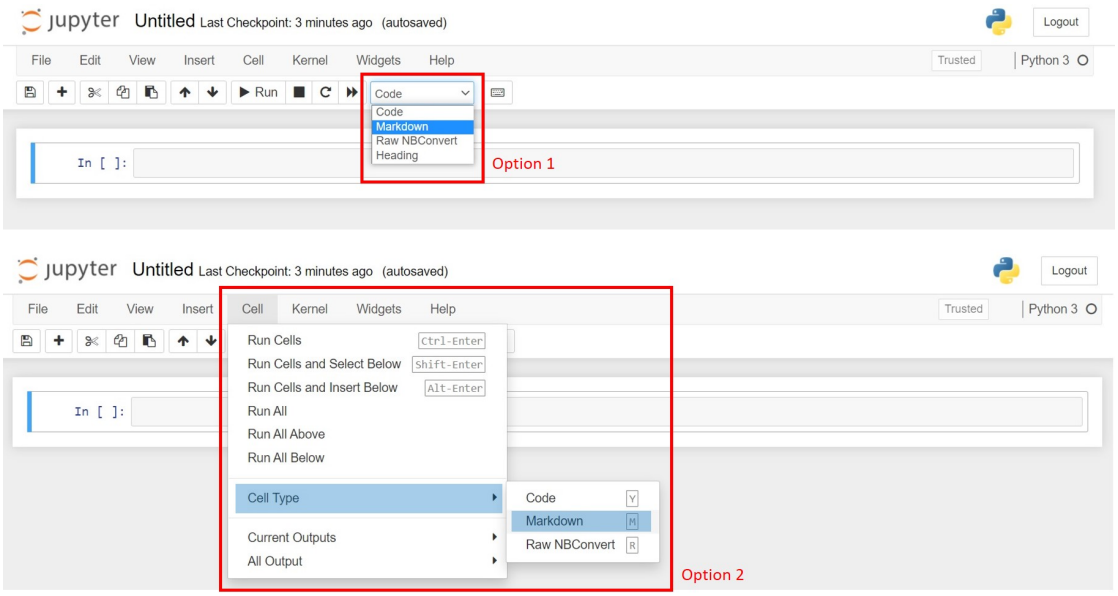


Figure 7: Two options on how to change the cell type to a "Markdown" option.

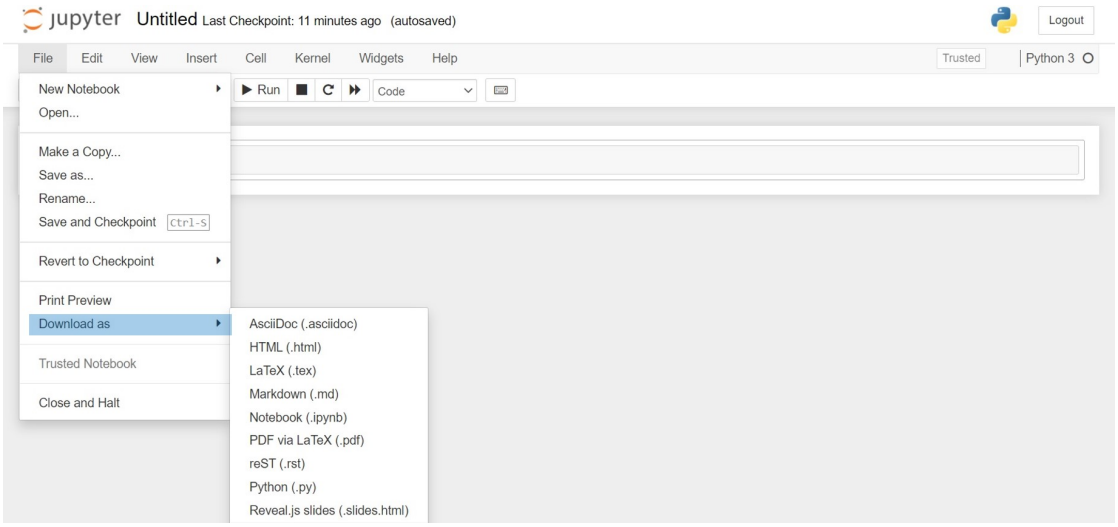


Figure 8: Downloading a Jupyter file as an another format.

3 Python Basics

3.1 Objects

Python contains many data types as part of the core language. As mentioned previously, Python is an object-oriented programming language, so it's important to understand that all data in a Python program is represented by objects and by relationships between objects. Objects whose value can change in the course of program execution are said to be mutable objects, whereas objects whose value is unchangeable after they've been created are called immutable. Each object in Python has three characteristics. These characteristics are:

- **object type**: Object type tells Python what kind of an object it is dealing with. A type could be a number, or a string, or a list, or something else.
- **object value**: Object value is the data value that is contained by the object. This could be a specific number, for example.
- **object identity**: you can think of object identity as an identity number for the object. Each distinct object in the computer's memory will have its own identity number.

Most Python objects have either data or functions or both associated with them. These are known as attributes. The name of the attribute follows the name of the object. And these two are separated by a dot in between them. The two types of attributes are called either data attributes or methods. A data attribute is a value that is attached to a specific object. In contrast, a method is a function that is attached to an object. And typically a method performs some function or some operation on that object. Object type always determines the kind of operations that it supports. In other words, depending on the type of the object, different methods may be available to you as a programmer.

3.2 Modules and Methods

Python modules are libraries of code that can be implemented using the `import` statements. Each module comes with several functions (or methods) associated with it. Sometimes, we don't want to use the entire module. Perhaps we just want to choose one function from that module. Let's think about a situation where I just need to be able to have the value of π available to me in my program. So, I don't need anything else from the imported library, just the value of π. To do that, I can tell Python from **math** library to import just π, as illustrated in Fig(9).

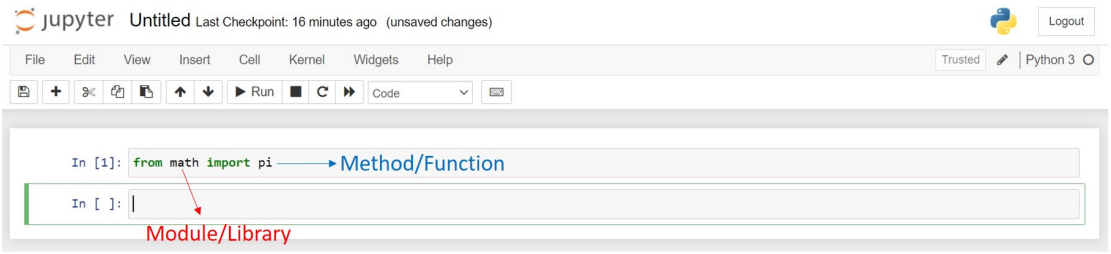


Figure 9: An Example of importing a method from a module.

It's usually helpful to know what are the available methods included in a given module. To list all the module's functions, we can use the `dir(module)` command to show the most relevant methods available.

4 Scalars, Vectors and Matrices

Linear algebra is one of the fundamental mathematical topics that provides a solid understanding to many science and engineering systems. Scalars, vectors and matrices are the basic elements used in linear algebra, that form the backbone of various methods, equations and algorithms. Hence, it is crucial for scientists and engineers to understand these core ideas. In this section, we will go through the definitions of these key elements and how to define them in Python through examples.

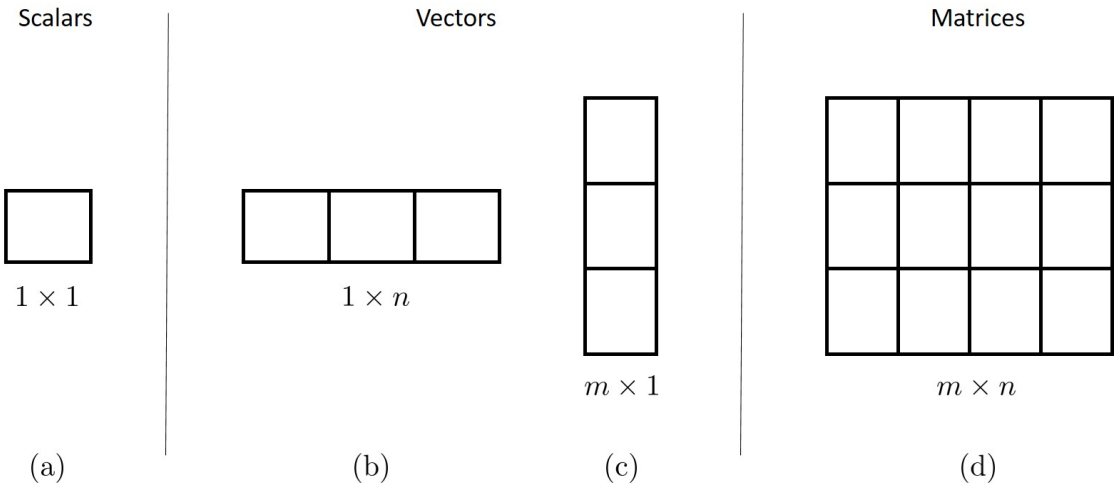


Figure 10: Simplified sketch explaining the difference between scalars, vectors and matrices.

4.1 Scalars

A scalar is a single number. Normally, a scalar represents a magnitude of a quantity. For example temperature, distance, speed, or mass – all these quantities have a magnitude and can be defined as a single number. Numbers are one type of objects in Python. In fact, Python provides three different numeric types: integers, floating point numbers, and complex numbers. Basic arithmetic operations can be applied to all of these scalars types, e.g., addition, subtraction, division, multiplication, .. etc. Fig(11) shows how to define different types of scalar numbers, and the main operations applied on them.

```
In [32]: #Integers and float numbers
         x = 2 #Define an integer
         y = 2.5 #Define a float
         #Basic Operations
         display(x+y)  #Addition
         display(y-x)  #Subtraction
         display(x*y)  #Multiplication
         display(x/y)  #Division
         display(x//y) #Floor division
         display(x**y) #Exponentiation

         #Importing "cmath" for complex number operations
         import cmath
         # Initializing real numbers
         x = 5
         y = 3
         # converting x and y into complex number
         z = complex(x,y);
         # display the real and imaginary part of complex number
         display(z.real)
         display(z.imag)
```

Figure 11: Defining different types of scalars and the main arithmetic operations.

4.2 Vectors

A vector is an array of numbers. The numbers are arranged in order and we can identify each individual number by its index in that ordering. In simple terms, a vector is an arrow representing a quantity that has both magnitude and direction wherein the length of the arrow represents the magnitude and the orientation tells you the direction. For example wind, which has a direction and magnitude. Vectors can be structured either as a row vector, as shown in Fig(10,b), or a column vector, Fig(10,c). Row vectors have n columns with only one single row, and this can be written as $1 \times n$. Whereas, column vectors structure the data vertically where they have m rows with only one column, and this can be written as $m \times 1$. Whether to use row or column vectors depends mainly on the context and the studied problem.

In Python, there are a number of ways to create a row and column vector, as shown in Fig(12) and Fig(13). Notice, that when defining a column vector, each row is included in a square brackets separated by a comma from the other rows.

```
In [1]: #Import numpy package
        import numpy as np
```

```
In [12]: #Equivalent ways to create a row vector
         x1 = np.array([1,2,3])
         x2 = np.matrix([1,2,3])

         display(x1) #show vector x1
         display(x2) #show vector x2
```

Figure 12: Different ways to create a row vector.

There are other ways to generate vectors. Let say, you want to generate a vector

```
In [13]: #Equivalent ways to create a column vector
         y1 = np.array([[1],[2],[3]]) # notice that each row is in square brackets
         y2 = np.matrix([[1],[2],[3]])

         display(y1) #show vector y2
         display(y2) #show vector y3
```

Figure 13: Different ways to create a column vector.

within a given interval. This can be easily done using the `arange()` method, as shown in Fig(14). Very important here to mention is that vectors in Python are zero-indexed, meaning that Python starts indexing (or counting) rows' and columns' elements from 0, i.e., 0^{th} element, 1^{st} element, 2^{nd} element, ...onwards. Therefore, when running the code in Fig(14), the last displayed element of the vector will be before the one specified in the code.

```
In [14]: #Create a vector within a range [a,b], where a is the start of the range, and b is the end+1.
         z1 = np.arange(0,10) #0 is the start, and 10 is the end+1
         z2 = np.arange(0,10,1) #0 is the start, 10 is the end+1, 1 is the step taken between each value.
         z3 = np.arange(0,9,2) #0 is the start, 9 is the end+1, 2 is the step taken between each value.

         display(z1) #show vector z1
         display(z2) #show vector z2
         display(z3) #show vector z3
```

Figure 14: Define a vector within a given range.

4.3 Matrices

A matrix is a 2D array. In other words, a matrix is a collection of m rows (its height) and n columns (its width) of data, as shown in Fig(10,d). The matrix dimension is usually written as $m \times n$, so each element is identified by two indices instead of just one. You can notice that vectors can be described as special cases of matrices. For example, a row vector is a matrix with only one row. Similarly, a column vector is a matrix with only one single column.

To generate a matrix in Python, the `array` method from **numpy** library is used, as shown in Fig(15).

```
In [17]: #Create a 3x3 matrix: 3 rows and 3 columns
         A = np.array([[1,2,3],[4,5,6],[7,8,9]])

         display(A) #show matrix A
```

Figure 15: Create a matrix.

Once the matrix is created, there are various ways to manipulate it and pull out information from it that could be useful for the computation that you need to

do. This can be done by accessing its main components, i.e., rows, columns and elements, as illustrated in Fig(16). As mentioned previously, always remember that Python is a zero-indexed language, meaning that it starts indexing (or counting) rows and columns of a matrix from 0.

```
In [19]: #Accessing and manipulating matrix rows, columns and elements
         a1 = A[1,2] #access the element in the 2nd row and 3rd column
         a2 = A[1,:] #access 2nd row with all its elements/columns (the colon (:) means access everything in that row)
         a3 = A[:,2] #access 3rd column with all its elements/rows (the colon (:) means access everything in that column)

         display(a1) #show a1
         display(a2) #show a2
         display(a3) #show a3
```

Figure 16: Accessing matrix elements.

When dealing with matrices, it is essential to first describe its shape, size and dimension to understand their main features, as Fig(17) shows.

```
In [29]: #Describing matrices
         display(A.shape) #show the number of rows and columns

         display(A.size) #show the number of elements (rows*columns)

         display(A.ndim) #show the number of dimensions
```

Figure 17: Different ways to describe a matrix.

Matrices have basic operations that are necessary to apply in many problems, e.g., addition, subtraction, and transpose. Transpose is a common operation in linear algebra where the column and row indices of each element are swapped. These operations have straightforward commands in Python to implement, as shown in Fig(18).

```
In [25]: #Matrix operations
         A = np.array([[1,2,3],[4,5,6],[7,8,9]]) #Define matrix A
         B = np.array([[10,11,12],[13,14,15],[16,17,28]]) #Define matrix B

         #Addition
         Add1 = A+B
         Add2 = np.add(A,B)

         display(Add1)
         display(Add2)

         #Subtraction
         Sub1 = A-B
         Sub2 = np.subtract(A,B)

         display(Sub1)
         display(Sub2)

         #Transpose
         AT = A.T #Take the transpose of matrix A
         BT = B.T #Take the transpose of matrix B

         display(AT)
         display(BT)
```

Figure 18: Matrix Operations.

4.4 Problem: Work

Work (W (J)) is the transfer of energy by a force ($\vec{\mathbf{F}}$ (N)) acting on an object as it is moved from one place to another- that is, undergoes a displacement ($\vec{\mathbf{s}}$ (m)). You do more work if the force or the displacement is greater. Mathematically, we define the work done by a constant force as the dot product between the force and displacement vectors:

$$W = \vec{\mathbf{F}} \cdot \vec{\mathbf{s}} \tag{1}$$

An essential point to mention here is that work is a scalar quantity, even though it's calculated from two vector quantities. Notice that this equation has a form of the scalar product of two vectors:

$$\vec{\mathbf{A}} \cdot \vec{\mathbf{B}} = |\vec{\mathbf{A}}||\vec{\mathbf{B}}| \cos\phi \qquad \text{and} \qquad \vec{\mathbf{A}} \cdot \vec{\mathbf{B}} = A_x B_x + A_y B_y + A_z B_z$$

where: $\vec{\mathbf{A}} = A_x\hat{i} + A_y\hat{j} + A_z\hat{k}$ and $\vec{\mathbf{B}} = B_x\hat{i} + B_y\hat{j} + B_z\hat{k}$. Therefore, it can be re-written as:

$$\vec{\mathbf{F}} \cdot \vec{\mathbf{s}} = |\vec{\mathbf{F}}||\vec{\mathbf{s}}| \cos\phi \qquad \text{and} \qquad \vec{\mathbf{F}} \cdot \vec{\mathbf{s}} = F_x s_x + F_y s_y + F_z s_z \tag{2}$$

where $|\vec{\mathbf{F}}|$ and $|\vec{\mathbf{s}}|$ are the magnitudes of the force and displacement vectors, respectively, and Φ is the angle between the force vector and displacement direction.

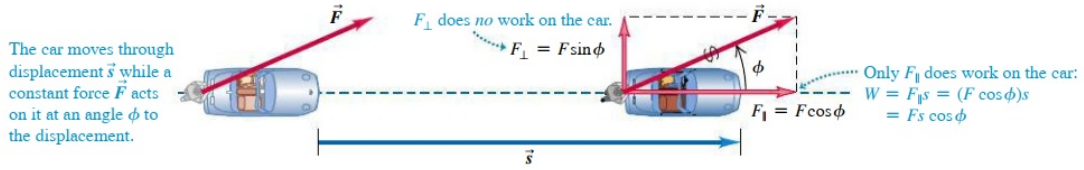


Figure 19: The work done by a constant force acting at an angle to the displacement [6].

As an illustration of this equation [6], think of a person exerting a force on a stalled car of magnitude 210 N in Fig(19), as he pushes it in a distance of 18 m. The car has also a flat tire, so to make the car track straight, the person needs to push at an angle of 30^o to the direction of the motion. To calculate the work here, and since we have information of the magnitudes of the force and displacement with the angle between them, we can need use Eq(2) directly, as follows:

$$\begin{aligned} W = \vec{\mathbf{F}} \cdot \vec{\mathbf{s}} &= |\vec{\mathbf{F}}||\vec{\mathbf{s}}| \cos\phi \\ &= (210\text{N})(18\text{m})\cos(30^o) \\ &= 3.3 \times 10^3\text{J} \end{aligned}$$

In Python, this can be calculated as shown in Fig(20). Notice the use of the **numpy** library to call both methods: π and cos.

```
In [52]: #import numpy library
         import numpy as np

         #Define input values
         F = 210 # force magnitude in N
         s = 18 # displacement magnitude in m
         #phi = (np.pi)/6 #angle in radians between the force and the displacement vectors (30 deg)
         phi = 30*(np.pi/180) #or transform the angle from degrees to radians

         #Calculate work
         W = F*s*(np.cos(phi)) #in J
```

Figure 20: Calculating work using the magnitudes of force $|\vec{\mathbf{F}}|$ and displacement $|\vec{\mathbf{s}}|$ with the angle between their vectors ϕ.

This person needs to push a second stalled car with a steady force $\vec{\mathbf{F}} = (160\text{N})\hat{i} - (40\text{N})\hat{j}$. The displacement of the car is $\vec{\mathbf{s}} = (14\text{m})\hat{i} + (11\text{m})\hat{j}$. To compute the work here, we notice that we have been provided by the components of the force and displacement vectors. Using Eq(2), we can write the following:

$$\begin{aligned} W = \vec{\mathbf{F}} \cdot \vec{\mathbf{s}} &= F_x s_x + F_y s_y \\ &= (160\text{N})(14\text{m}) + (-40m\text{N})(11\text{m}) \\ &= 1.8 \times 10^3 \text{J} \end{aligned}$$

In Python, this can be computed as shown in Fig(21). Notice how arrays (row vectors) are defined using **numpy** library and how their elements are called.

```
In [55]: #import numpy library
         import numpy as np

         #Define the vectors
         F = np.array([160 , -40]) #Force vector
         s = np.array([14 , 11]) #displacement vector

         #Calculate the sum of the products of their respective components
         W = F[0]*s[0] + F[1]*s[1] #in J
```

Figure 21: Calculating work using the components of the force $\vec{\mathbf{F}}$ and displacement $\vec{\mathbf{s}}$ vectors.

Now, suppose we have the following two vectors:

$$\vec{\mathbf{F}} = 2\hat{i} + 3\hat{j} + 1\hat{k} \qquad \text{and} \qquad \vec{\mathbf{s}} = -4\hat{i} + 2\hat{j} - 1\hat{k}$$

and we need to find the angle between both of them. To do that, we solve Eq(2) for $\cos\phi$:

$$\begin{aligned}\cos\phi = \frac{\vec{\mathbf{F}}\cdot\vec{\mathbf{s}}}{|\vec{\mathbf{F}}||\vec{\mathbf{s}}|} &= \frac{F_x s_x + F_y s_y + F_z s_z}{\sqrt{F_x^2+F_y^2+F_z^2}\sqrt{s_x^2+s_y^2+s_z^2}} \\ &= \frac{(2)(-4)+(3)(2)+(1)(-1)}{\sqrt{(2)^2+(3)^2+(1)^2}\sqrt{(-4)^2+(2)^2+(-1)^2}} \\ &= \frac{-3}{\sqrt{14}\sqrt{21}} \\ &= -0.175\end{aligned}$$

Taking the inverse of the cosine gives $\phi = 100^o$. Notice here that the computed work is negative and the angle is between 90^o and 180^o. What does that mean? Fig(22) shows the code to compute this using Python.

In [58]:
```python
#import numpy
import numpy as np

#Define the vectors
F = np.array([2,3,1])
s = np.array([-4,2,-1])

#Calculate the sum of the products of their respective components
FTimess = F[0]*s[0] + F[1]*s[1] + F[2]*s[2]

#Calculate their magnitudes
#Vector F
Fsquare = np.square(F) #the square of each component of vector F
Fsum    = np.sum(Fsquare) #the sum of the squared components of F
Fmag    = np.sqrt(Fsum) #square root of the sum
#Vector s
ssquare = np.square(s) #the square of each component of vector s
ssum    = np.sum(ssquare) #the sum of the squared components of s
smag    = np.sqrt(ssum) #square root of the sum

#Calculate the cosine of the angle
CosinePhi = FTimess / (Fmag * smag)

#Calculate the angle (inverse of the cosine)
Phi_rad = np.arccos(CosinePhi) #in radians
Phi_deg = Phi_rad*(180/np.pi) #in degrees

#We notice here that the work is negative and the angle is between 90 and 180. This means ...
```

Figure 22: Calculating the scalar product of two vectors: $\vec{\mathbf{A}}$ and $\vec{\mathbf{B}}$.

5 Expressions and Booleans

Operators in Python are used to perform operations on variables and values. In this section, we will cover three main families of operators used in Python: comparison, logical, and identity operators, listed in in Table(26).

Expression is a combination of objects and operators that computes a value. Many expressions involve what is known as the boolean data type. Objects of the boolean type have only two values: `True` or `False`. In Python, you need to capitalize these words for Python to understand them as a boolean type. Note from Fig(23) that not capitalising "`False`" results an error as Python doesn't know what this object is.

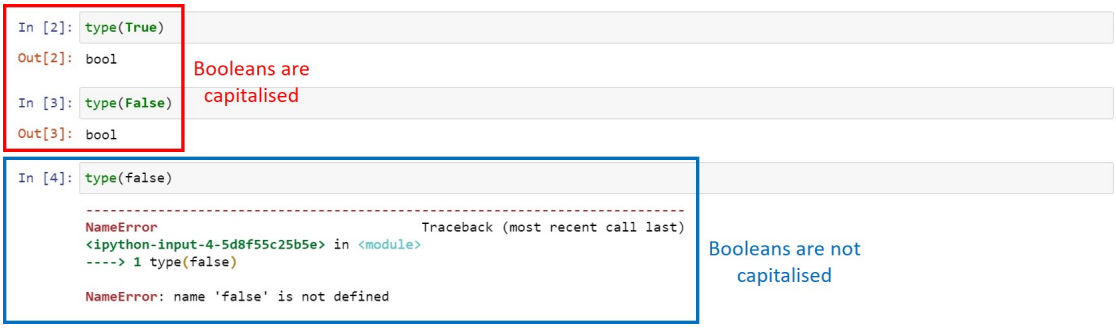


Figure 23: Booleans in Python.

The simplest case where boolean data type occurs is when two objects are compared. We often need to compare objects in our programs. There are a total of six different comparison operations in Python, listed in Table(1). Although these are commonly used for numeric types, we can actually apply them to other types as well. The result of a comparison operation is always a boolean type, either `True` or `False`. An example of using these operators is shown in Fig(24).

Operations involving logic, so-called logical (or boolean) operations, take in one or more boolean object and then they return one boolean object back to you. There are only three logical operators:"`or`", "`and`", and "`not`".

Let's start with "`or`". "`or`" between `x` and `y` is going to be `True` if either `x` is `True` or `y` is `True`, or both are `True`. So for example, if we say `True or False`, then Python returns `True`. `True or True` would also be `True`. So the only time "`or`" would be `False`, if both the first and second object surrounding "`or`" are `False`.

On the other hand, "`and`" is only `True` if both objects are `True`. So, if we type `True and True`, the answer is going to be `True`. However, if we turned the second `True` to `False`, "`and`" is going to be `False`. So, in order for "`and`" to be `True`, both of the objects need to be `True`.

```
In [5]: x = 10
        y = 12

        # Output: x > y is False
        print('x > y is',x>y)

        # Output: x < y is True
        print('x < y is',x<y)

        # Output: x == y is False
        print('x == y is',x==y)

        # Output: x != y is True
        print('x != y is',x!=y)

        # Output: x >= y is False
        print('x >= y is',x>=y)

        # Output: x <= y is True
        print('x <= y is',x<=y)
```

Figure 24: Example of Comparison Operators in Python.

Finally, we have the "`not`" operator, which simply negates the value of the object. So if we say `not True`, Python gives us `False`. And if we say `not False`, Python returns to us `True`. Fig(25) shows an example illustrating this.

```
In [6]: x = True
        y = False

        # Output: x and y is False
        print('x and y is',x and y)

        # Output: x or y is True
        print('x or y is',x or y)

        # Output: not x is False
        print('not x is',not x)
```

Figure 25: Example of Logical Operators in Python.

The third family of operators that we need to cover is the identity operators. Identity operators in Python are "`is`" and "`is not`". They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical. Notice that this is different than asking if the contents of two objects are the same.

The example in Fig(26) shows that `x1` and `y1` are integers of the same values, so they are equal as well as identical. Same is the case with the strings `x2` and `y2`. But `x3` and `y3` are lists. They are equal but not identical. It is because the interpreter locates them separately in memory although they are equal. So, although these two lists have the same content, but they are two individual and distinct objects. That's why this comparison returns `False`.

```
In [7]: x1 = 5
        y1 = 5
        x2 = 'Hello'
        y2 = 'Hello'
        x3 = [1,2,3]
        y3 = [1,2,3]

        # Output: False
        print(x1 is not y1)

        # Output: True
        print(x2 is y2)

        # Output: False
        print(x3 is y3)
```

Figure 26: Example of Identity Operators in Python.

Operator	Name	Example
Comparison Operators		
==	Equal	$x == y$
! =	Not equal	$x! = y$
>	Greater than	$x > y$
<	Less than	$x < y$
>=	Greater than or equal to	$x >= y$
<=	Less than or equal to	$x <= y$
Logical Operators		
and	Returns True if both statements are true	$x < 5$ and $x < 10$
or	Returns True if one of the statements is true	$x < 5$ and $x < 4$
not	Reverse the result, returns False if the result is true	not ($x < 5$ and $x < 10$)
Identity Operators		
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Table 1: Python Operators

6 Iterative Loops

Loops are essential in almost every programming language. Loops are among the most basic and powerful of programming concepts. A loop in a computer program is an instruction that repeats until a specified condition is reached. In a loop structure, the loop asks a question. If the answer requires an action, it is executed. The same question is asked again and again until no further action is required. Each time the question is asked is called an iteration. Just about every programming language includes the concept of a loop. One of the main types of loops is the **for** loop.

6.1 for Loop

for Loop repeats a block of code a set number of times. The reason they are called **for** loops is that you can tell your program how many times you want it to repeat the code for. **for** loops help you reduce repetition in your code because they let you execute the same operation multiple times. **for** loops use a variable to count how many times the code has been repeated, called a counter. You control how many times the loop repeats by setting where the counter starts and ends. You also set how much the counter goes up by each time the code repeats. In most scenarios, you'll want the counter to increase by 1 each time the loop repeats. The syntax for **for** loop can be shown as following:

Here, the `sequence` can be either a list of strings or a range of numbers. The `item`, on the other hand, is the variable that iterate over the elements of the list, and every time it is iterated, the line (or multiple lines) of code is executed. It is very important here to mention that you need to be careful with the syntax. At the end of the for line, a colon is required. Also, Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. This means that lines that contain expressions to execute must be intended.

For example, Fig(27) shows how to compute the squared values of each number in a given list using **for** loop. "**for**" and "**in**" are Python keywords and `numbers` is the name of our list. Also, "`i`" is a temporary variable and its only role is to store the given element of the list that we will work with in the given iteration of the loop. Even if this variable is called `i` most of the time (in online tutorials or books for instance), it's good to know that the naming is totally arbitrary. The

function inside the loop asks the loop to print the result if the squared value each time an iteration is achieved.

```
In [1]: #Create a list of numbers
        numbers = [1, 5, 12, 91, 102]

        #print the result of the square of each element
        for i in numbers:
            print(i * i)
```

Figure 27: **for** loop over a list of numbers.

As mentioned before, you can use other sequences than lists too. For example a string, as shown in Fig(28). Remember, strings are basically handled as sequences of characters, thus the **for** loop will work with them pretty much as it did with lists.

```
In [2]: #Create a list of letters
        letters = "Hello World!"

        #print the characters one by one
        for i in letters:
            print(i)
```

Figure 28: **for** loop over a list of characters.

An important function should be mentioned in this context is the `range()` function. `range()` is a built-in function in Python andit is used almost exclusively within **for** loops. It mainly generates a list of numbers. As shown in Fig(29), it accepts three arguments:

- **The first element:** this will be the first element of your range.
- **The last element:** you might assume that this will be the last element of your range... but it isn't. Again, as mentioned previously, Python is zero-indexed, which means that it starts counting from zero. Therefore, if the last element is 10, as in the example shown, then the range will go from 0 to 9.
- **The step:** this is the difference between each element in the range. So if it's 2, you will only print every second element.

Note: the first element and the step attributes are optional. If you don't specify them, then the first element will be 0 and the step will be 1 by default.

```
In [3]: #Create a range of numbers
        x = range(0,10) #Remember, Python is zero-indexed. Hence, the range starts from 0 and ends at 9 (and not 10). The step here is 1
        #x = range(0,10,2) the list here starts from 0 and goes with step 2 all the way to 9

        #print the numbers within the given range
        for i in x:
            print(i)
```

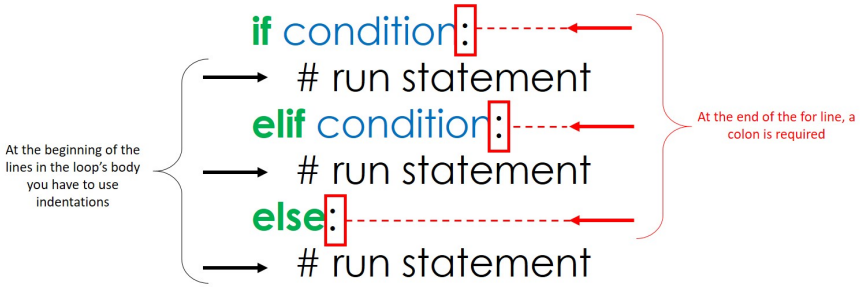
Figure 29: **for** loop over a range of numbers.

7 Conditional Statements

Conditionals (that is, conditional statements, conditional expressions and conditional constructs) are programming language commands for handling decisions. Specifically, conditionals perform different computations or actions depending on whether a programmer-defined boolean condition evaluates to true or false. **if** statement is a key part of the decision-making process in programming.

7.1 If Statement

The **if** statement is one of the powerful conditional statement and is common across many programming languages. It is responsible for modifying the flow of execution of a program. It is always used with a condition. The condition is evaluated first to either true or false before executing any statement inside the body of **if**. The syntax for **if** statement can be broadly constructed as follows:



Similar to **for** loop, the **if** statement requires a colon at the end of the **if**, **elif** and **else** first lines. Moreover, as mentioned before, Python relies on indentation to define scope in the code, and that's also applied to **if** statement structure. If no indentation is applied, you will get an error.

It is noticed from the structure that there are three keywords that are used in the **if** statement: **if**, **elif** and **else**. Both, **if** and **elif** are followed by a condition to be evaluated. These conditions are mainly formulated using one of operators that Python supports, listed in Table(1). Some codes do not require for **elif** and else to be in the structure of **if** statement. Sometimes, **if** is enough to achieve the required task. For example, Fig(30) shows that we can evaluate the comparison between `a` and `b` just by using the **if** keyword. In this example we use two variables, `a` and `b`, which are used as part of the **if** statement to test whether `b` is greater than `a`. As `a` is 33, and `b` is 200, we know that 200 is greater than 33, and so we print to screen that "`b is greater than a`".

The **elif** keyword is python's way of saying "if the previous conditions were not true, then try this condition". As shown in the example in Fig(31), `a` is equal

```
In [2]: #Define two different numbers
        a = 33
        b = 200

        #Compare both numbers using if statement
        if b > a:
          print("b is greater than a") #if the condition returns "True", execute this line
```

Figure 30: The use of the **if** keyword in **if** statement.

to `b`, so the first condition is not true, but the **elif** condition is true, so we print to screen that "`a and b are equal`".

```
In [3]: #Define two different numbers
        a = 33
        b = 33

        if b > a: #first conditon
          print("b is greater than a")

        elif a == b: #if the previous condition was not true, try this condition
          print("a and b are equal")
```

Figure 31: The use of the **if** and **elif** keywords in **if** statement.

The **else** keyword, on the other hand, catches anything which isn't caught by the preceding conditions. In the example in Fig(32), `a` is greater than `b`, so the first condition is not true, also the **elif** condition is not true, so we go to the **else** condition and print to screen that "`a is greater than b`".

```
In [4]: #Define two different numbers
        a = 200
        b = 33

        if b > a: #first conditon
          print("b is greater than a")
        elif a == b: #if the previous condition was not true, try this condition
          print("a and b are equal")
        else: #if non of the previous conditions is true, execute this code
          print("a is greater than b")
```

Figure 32: The use of the **if**, **elif** and **else** keywords in **if** statement.

You can also have an **else** without the **elif** within the **if** structure, as shown in the example in Fig(33).

```
In [5]: #Define two different numbers
        a = 200
        b = 33

        if b > a: #first conditon
          print("b is greater than a")
        else: #if the first conditoon is not true, execute this code.
          print("b is not greater than a")
```

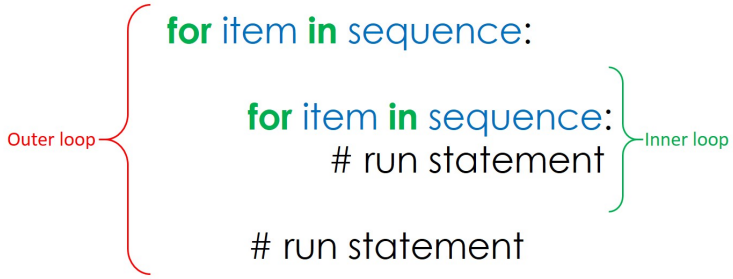
Figure 33: The use of the **if** and **else** keywords in **if** statement.

8 Nesting

Nesting occurs when one programming construct is included within another. Nesting allows for powerful, yet simple programming. It reduces the amount of code needed, while making it simple for a programmer to debug and edit. In the following, we will see how loops and conditions can be nested illustrated by examples.

8.1 Nested for

Nesting can applied on iterations, i.e., multiple layers of loops one within another. The main structure of such a nest can be viewed bellow:



It can be seen that, for the case of two layers of iterations, there are two nested loops to go through: the inner and the outer loops. For each iteration of an outer loop, the inner loop executes all its iterations before the outer loop can continue to its next iteration.

We should note here that the outer loop can contain more than one inner loop. There is no limitation on the chaining of loops.

In the nested loop, the number of iterations will be equal to the number of iterations in the outer loop multiplied by the iterations in the inner loop.

As shown in Fig(34), nested loops can be used to go over string lists. The outer **for** loop goes through each element of "`adj`" list and for each iteration the code asks the inner **for** loop to go over each element of "`fruit`" list and prints the combinations that results from each iteration.

Nested **for** loops can be also used over a range of numbers, as shown in Fig(35). It has been mentioned before that the `range()` command in Python goes all the way through the specified range minus the last number. The outer **for** loop shown in the code iterates from 1 to 10 (not 11!) and asks the inner loop to iterate through another range from 1 to 10 and prints the result of the product of each of their elements in each iteration.

```
In [26]: #Define two lists
         adj = ["red", "big", "tasty"]
         fruits = ["apple", "banana", "cherry"]

         #Nest of two loops
         #Outer loop
         #to iterate through adj list
         for x in adj:
             #inner loop
             #to iterate through fruits list
             for y in fruits:
                 #print all the combinations between adj and fruits lists
                 print(x, y)
```

Figure 34: Two nested **for** loops over string lists.

```
In [20]: # outer loop
         #iterate through all the numbers within a range from 1 to 10
         for i in range(1, 11):
             # inner loop
             # to iterate from 1 to 10
             for j in range(1, 11):
                 # print multiplication
                 print(i * j, end=' ')
             #print the value of each iteration of theo outer loop
             print()
```

Figure 35: Two nested **for** loops over two ranges of numbers.

The code shown in Fig(36) illustrates how the indentation affects the shape of the output. It shows different outputs for different nested **for** loop structures. For example in (a), the command `print()` resides outside the nested loops, therefore the result only shows the final output. However, in (b), and since the `print()` command sits inside the body of the inner loop, then it shows the result of each of its iteration.

```
In [24]: #Define two lists
         first = [2, 3, 4]
         second = [20, 30, 40]
         #initialise the list that will store the outputs
         final = []

         # outer loop
         #iterate through all the numbers in "first" list
         for i in first:
             # inner loop
             # to iterate through all the numbers in "second" list
             for j in second:
                 #calculate the sum of each element and put the result in "final" list
                 final.append(i+j)

         #print the output in "final" list
         print(final)

         [22, 32, 42, 23, 33, 43, 24, 34, 44]
```

(a)

```
In [25]: #Define two lists
         first = [2, 3, 4]
         second = [20, 30, 40]
         #initialise the list that will store the outputs
         final = []

         # outer loop
         #iterate through all the numbers in "first" list
         for i in first:
             # inner loop
             # to iterate through all the numbers in "second" list
             for j in second:
                 #calculate the sum of each element and put the result in "final" list
                 final.append(i+j)
             #print the output in "final" list
             print(final)

         [22, 32, 42]
         [22, 32, 42, 23, 33, 43]
         [22, 32, 42, 23, 33, 43, 24, 34, 44]
```

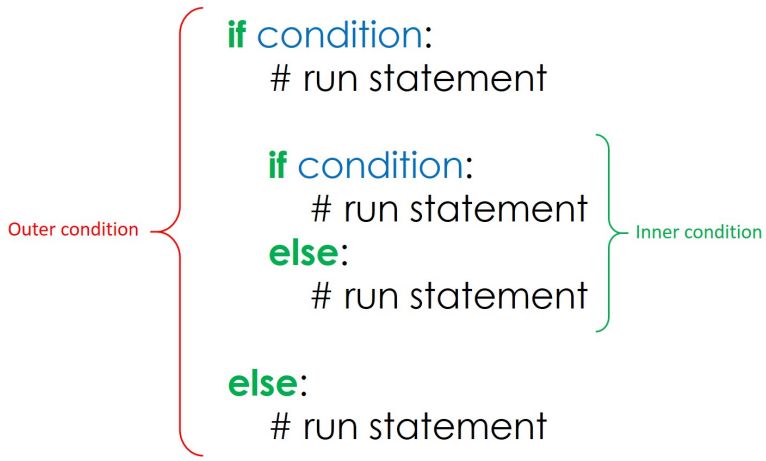
(b)

Figure 36: Two nested **for** loops over to lists of numbers and store the output in a different list.

8.2 Nested if

Similar to iterations, conditions can also be nested, i.e., you can have **if** statements inside **if** statements. There may be a situation when you want to check for another condition after a condition resolves to true. Any number of these conditions can be nested inside one another. Indentation is the only way to figure out the level of nesting. The general structure of nested conditions is shown below.

The code shown in Fig(37), asks the user to input a number (positive or negative). The outer **if** condition tests whether the number is positive, otherwise the **else** command prints on the screen that the number if negative. The inner condition, however, and provided the result of the outer condition is true, it tests whether the number is zero.




```
In [27]: #Ask the user to enter a number
         num = float(input("Enter a number: "))

         #Outer condition
         if num >= 0: #test if the number is positive
             if num == 0: #if the number is positive, test whether is equal to zero
                 print("Zero")
             else: #if the number is not zero, display that it's positive
                 print("Positive number")
         else: #if the result of the outer condition is false, print that the number is negative
             print("Negative number")
```

Figure 37: Two nested **if** conditions that test whether a given number is positive or negative.

8.3 Nested for and if

We can also have a hybrid combination of nested **for** loops and **if** conditions in the same structure. Again: when you use an **if** statement within a **for** loop, be extremely careful with the indentations because if you misplace them, you can get errors or incorrect results!

It's also worth mentioning here two common commands when using hybrid nesting: `break` and `continue`. In Fig(38), we have two programs, in (a), the code goes through all the elements of "`fruits`" list via for loop. In the inner **if** condition, it asks if the element is equal to "`banana`". If it's true, the code must `break`. Otherwise, it should print the result of the iteration. With the `break` statement, we stop the loop before it has looped through all the items. That's why the answer for this chunk of code is only `apple`. On the other hand, in (b), the code goes through the same "`fruits`" list via the for loop, but here when the **if** statement asks whether the iterative variable equals to "`banana`", it demands the code to `continue`. With the `continue` statement, we stop the current iteration of the loop, and continue with the next. That's why the result of this code is `apple` and `cherry`.

```
In [35]: fruits = ["apple", "banana", "cherry"]
         for x in fruits:
             if x == "banana":
                 break
             print(x)
```

(a)

```
apple
```

```
In [36]: fruits = ["apple", "banana", "cherry"]
         for x in fruits:
             if x == "banana":
                 continue
             print(x)
```

(b)

```
apple
cherry
```

Figure 38: The use of `break` and `continue` in hybrid nesting.

8.4 Problem: Bisection Method

To make a practical example of the use of **for** and **if** statements, we consider the bisection method for finding zeros of function [7]. In particular, we will consider the function:

$$f(x) = x^3 + 1 = 0 \tag{3}$$

for which the values of x which make this true must be found computationally. Fig(39) shows the function with the intersection point that represents its root (the red circle) over the interval $x \in [-4, 4]$ where $x_l = -4$ and $x_r = 4$.

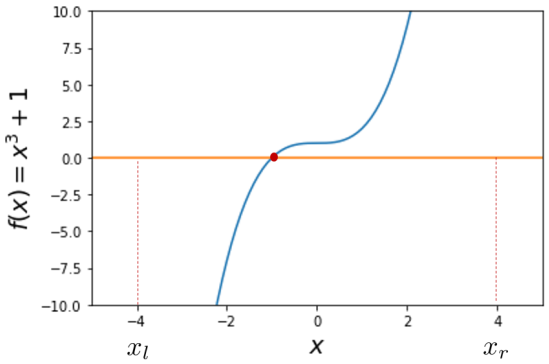


Figure 39: The function $f(x) = x^3 + 1$ with the corresponding zero (the red circle) over the interval $x \in [-4, 4]$ where $x_l = -4$ and $x_r = 4$.

The bisection method simply cuts the given interval in half and determines if the root is on the left or right side of the cut. Once this is established, a new interval is chosen with the midpoint now becoming the left or right end of the new domain, depending of course on the location of the root. This method is repeated until the interval has become so small and the function considered has

come to within some tolerance of zero. The following algorithm uses an outside **for** loop to continue cutting the interval in half while the embedded **if** statement determines the new interval of interest. A second **if** statement is used to ensure that once a certain tolerance has been achieved, i.e., the absolute value of the function $f(x) = x^3 + 1$ is less than 10^{-5}, then the iteration process is stopped.

```
In [76]: xr = -4 #Initial left boundary
         xl = 4 #Initial right boundary

         #bisection iterative method
         #construct iterations
         for j in range(0, 100):
             xc = (xr + xl)/2 #calculate the midpoint
             fc = xc**3+1 #calculate the function at xc
             #first embedded condition
             if ( fc > 0 ):
                 xl = xc #move left boundary
             else:
                 xr = xc #move right boundary
             #second embedded condition
             if ( abs(fc) < 1e-5 ): #calculate the tolerance
                 print('The root of the function is: ', xc)
                 print('The value of the fucntion is: ', fc)
                 print('The number of iterations is: ', j) #show how many iterations used to achieve the root
                 break #quit the loop

         The root of the function is:  -1.0
         The value of the fucntion is:  0.0
         The number of iterations is:  2
```

Figure 40: The Bisection Algorithm to fins the root of the function $f(x) = x^3 + 1$.

Note that the `break` command ejects you from the current loop. This effectively stops the iteration procedure for cutting the intervals in half. It can be seen from Fig(40), that only two iterations were necessary to reach the value of the root of the function $f(x)$. Surely, the number of iterations mainly depends on the initialisation of the assumed values of the left and right boundaries.

9 Functions

Functions are a set of instructions bundled together to achieve a specific outcome. Once a function is written, it can be called and reused multiple times within the code. This makes Functions a good alternative to having repeating blocks of code in a program which increases the re-usability of code.

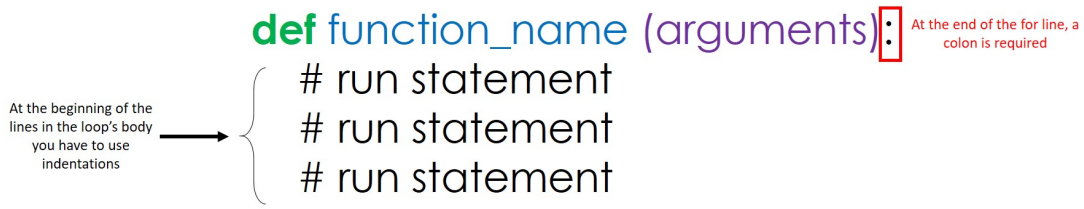
The main reason why we write functions is because they allow us to understand the program as sub-steps. Each sub-step can be its own function. When any program seems too hard, just break the overall program into sub-steps.

Common thing about functions is that they are famous with several names. Different programming languages name them differently, for example, functions, methods, sub-routines, procedures, etc.

The steps to write a clear and easy-to-understand function are:

- Understand the purpose of the function.
- Define the parameters that the function needs to accomplish its purpose.
- Decide on the set of steps that the program will use to accomplish this goal.
- Code, test, and optimize; go back to any previous step as necessary.

In Python, functions have certain syntax as shown below.



First, the keyword **def** marks the start of the function header and is used to define and create a function. A `function name` follows to uniquely identify the function. This name is essential when calling the function later in the code. Next, `parameters` (arguments) need to be defined within parentheses through which we pass values to a function. Remember, just like in loops and conditional statements, to end this line with a colon. The body of the function contains all the `statements` (instructions) that describe what the function does. Statements must have the same indentation level.

Let's clarify this with an example. Fig(41) shows a code that defines a function `myfun` which computes $y = x^3 + \sin x$. The first line of the function has three main

components: the key word **def**, its name `myfun`, and `x` as an argument. The body of the function has two statements to run, the first is the mathematical expression we need to compute, i.e., $y = x^3 + \sin x$, and the second is to let the function return the value of y. Once we have defined a function, we can simply call it by typing the function name with appropriate parameters. The code shows two ways to pass arguments through this function; either you pass only one value as in `out1`, or you define a range of values and you ask the function to compute the expression of y at each one of these values, as in `out2`. In python, the function definition should always be present before the function call. Otherwise, we will get an error.

```
In [3]: #importing numpy package
        import numpy as np

        #defining a function that computes x^3+sin(x)
        def myfun(x):
            y = x**3 + np.sin(x)
            return y

        # single input to the function
        out1 = myfun(1)
        display(out1)

        # vector of inputs to the function
        x = np.arange(0,11,1)
        out2 = myfun(x)
        display(out2)
```

Figure 41: A function that computes $y = x^3 + \sin x$.

Another example is shown in Fig(42). This code illustrates a function `funcA` that computes $f_A = \sin(Bx)$ where $B = A^2 + \cos(A)$ by passing more than one argument: two arguments to be specific; x and A. It is shown from the body of the function that two statements should be defined to achieve this computation. The first statement calculates B. The second statement uses the value of B obtained from before to compute the expression of f_A. The code ends by returning the values of both f_A and B. Once the structure of the function is finalised, it can be used and called later on in the code to compute the mathematical expressions for given arguments. In this code, it is shown that the function is called for the arguments $x = 3$ and $A = 2$.

```
In [3]: #importing numpy package
        import numpy as np

        def funcA(x,A):
            B = A**2 + np.cos(A)
            fA = np.sin(B*x)
            return fA,B

        # multiple inputs and outputs for a function
        out3 = funcA(3,2)
        display(out3)
```

Figure 42: A function that computes $f_A = \sin(Bx)$ where $B = A^2 + \cos(A)$.

Loops and conditions can be nested within functions as statements. The struc-

ture of both remains the same as explained before, but always pay attention to the indentation. Fig(43) shows a function `number` that goes through all the numbers of a given list `x` and test whether each number is a positive, negative or zero. To achieve this task, the argument of the function `x` should be a list of numbers. Next, a loop over all the elements of a given list `x` should be used and three conditions need to be implemented to test the three options on each element. Once the function is constructed, we can call it to apply it on list `A` and print out the type of all of its elements.

```
In [12]: def number(x):
             for num in x:
                 if num >= 0:
                     if num == 0:
                         print("Zero")
                     else:
                         print("Positive number")
                 else:
                     print("Negative number")

         #define a list of number A
         A = [4,5,-6,7,0]
         #call function "number" and test the numbers in list A
         out4 = number(A)
```

Figure 43: A function that evaluates each number of a list whether it's positive, negative or zero.

9.1 Problem: Simple Harmonic Motion

Many kinds of motion repeat themselves over and over: the vibration of a quartz crystal in a watch, the swinging pendulum of a grandfather clock, the sound vibrations produced by a clarinet or an organ pipe, and the back and forth motion of the pistons in a car engine. This kind of motion is called a periodic motion or oscillation [6]. A body that undergoes periodic motion always has a stable equilibrium position. When it is moved away from this position and released, a force or torque comes into play to put it back toward equilibrium. But by the time it gets there, it has picked up some kinetic energy, so it overshoots, stopping somewhere on the other side, and is again pulled back toward equilibrium. Fig(44) shows one of the simplest systems that can have periodic motion. A body with mass m rests on a frictionless horizontal guide system, such as a linear air track, so it can move along the x-axis only. The body is attached to a spring of negligible mass that can be either stretched or compressed. The left end of the spring is held fixed, and the right end is attached to the body. The spring force is the only horizontal force acting on the body; the vertical normal and gravitational forces always add to zero.

The simplest kind of oscillation occurs when the restoring force F_x is directly proportional to the displacement from equilibrium x. This happens if the spring

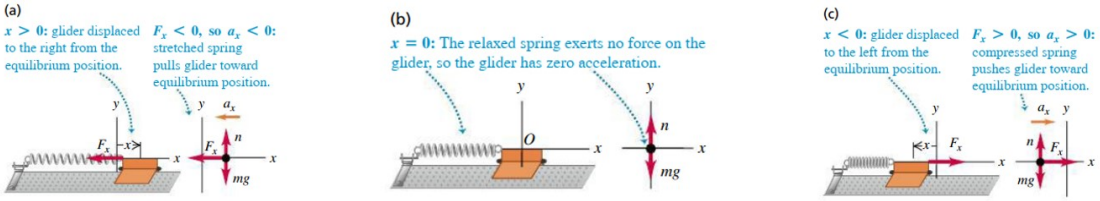


Figure 44: Model for periodic motion. When the body is displaced from its equilibrium position at $x = 0$, the spring exerts a restoring force back toward the equilibrium position [6].

is an ideal one and obeys *Hooke's law*. The constant of proportionality between F_x and x is the spring constant k and these quantities are related as:

$$F_x = -kx \tag{4}$$

This equation gives the correct magnitude and sign of the force, whether x is positive, negative, or zero. The spring constant k is always positive and has units of (N/m). The oscillation modelled by this equation is called *simple harmonic motion (SHM)*. This motion can be further understood by defining several other physical quantities as shown in Table(2).

Quantity	Equation	Definitions
Restoring Force	$F_x = -kx$	
Acceleration	$a_x = -\frac{k}{m}x$	m is the mass
Angular Frequency	$\omega = \sqrt{\frac{k}{m}}$	
Frequency	$f = \frac{\omega}{2\pi} = \frac{1}{2\pi}\sqrt{\frac{k}{m}}$	
Period	$T = \frac{1}{2f} = \frac{2\pi}{\omega} = 2\pi\sqrt{\frac{m}{k}}$	
Kinetic energy	$E_k = \frac{1}{2}mv^2$	v is the velocity
Potential energy	$E_p = \frac{1}{2}kx^2$	
Total mechanical energy	$E_t = E_k + E_p = \frac{1}{2}kA^2$	A is the amplitude

Table 2: Main physical quantities of simple harmonic motion.

Our aim now is to write a Python function that computes all of the physical quantities listed in Table(2) by inserting only the spring constant k, displacement

x, velocity v and mass m as inputs. Fig(45) shows the code that does this computation for several mass values and outputs the results as lists. This is informative as it tells us what happens to these quantities as the attached mass changes its value.

```
In [41]: #import numpy library
         import numpy as np

         #define the function to compute SHM main quantities for sevral masses
         def SHM(k,x,m,v):
             #Define the final arrays for all quantities
             Fx = list()
             ax = list()
             w  = list()
             f  = list()
             T  = list()
             Ek = list()
             Ep = list()
             Et = list()
             #Go over all the values of masses and calculate quanitties for each one
             for i in range(0,len(m)):
                 Fx1 = -k*x            #Restoring force
                 Fx.append(Fx1)        #keep this value and add it to the final list
                 ax1 = -(k/m[i])*x     #Acceleration
                 ax.append(ax1)        #keep this value and add it to the final list
                 w1  = np.sqrt(k/m[i]) #Angular frequency
                 w.append(w1)          #keep this value and add it to the final list
                 f1  = w1/2*(np.pi)    #Frequency. This can aslo be computed using w = (1/(2*np.pi))*(np.sqrt(k/m))
                 f.append(f1)          #keep this value and add it to the final list
                 T1  = 1/f1            #Period. This can also be computed using T = (2*np.pi)/w or T = 2*(np.pi)*(np.sqrt(m/k))
                 T.append(T1)          #keep this value and add it to the final list
                 Ek1 = (1/2)*m[i]*v**2 #Kinetic energy
                 Ek.append(Ek1)        #keep this value and add it to the final list
                 Ep1 = (1/2)*k*x**2    #Potential energy
                 Ep.append(Ep1)        #keep this value and add it to the final list
                 Et1 = Ek1 + Ep1       #Total Mechanical energy. This can aslo be computed using Et = (1/2)*k*A^2
                 Et.append(Et1)        #keep this value and add it to the final list
             #return all the final lists
             return Fx, ax, w, f, T, Ek, Ep, Et

         #Define the values you want to use for calculation
         m = [0.2, 0.3, 0.4, 0.5] #kg
         k = 200   #N/m
         x = 0.015 #m
         v = 0.40  #m.sec^-2

         #Call the function
         out = SHM(k,x,m,v)

         #disaply the results
         print(out)
```

Figure 45: Python function that calculates several physical quantities of the simple harmonic motion: restoring force, acceleration, angular frequency, frequency, period, kinetic energy, potential energy and the total mechanical energy, given only the mass, displacement, spring constant and the velocity.

Notice the use of the `append()` method. The `append()` method appends an element to the end of the list. This method allows keeping the results of each iteration and assign them to the pre-assigned lists. Hence, the lists change their sizes in each iteration. If this method is not used, Python only shows the results of the last calculation throwing away the previous ones.

10 Working with Data

Today, we're living in a data-driven world with the availability of a vast and increasing quantities of data in all sectors of life. Analysing this data is a fundamental key to effectively understand it and identify trends, patterns or even abnormalities in its behaviour. This helps in interpreting important insights and making right decisions.

Data analysis has a long list of tools that help scientists and engineers get a sense of the data they have and build on it. Python is one of the most common tools most academics use to easily load, understand, clean, interpret, and visualise the dynamics of the data. In the following, we will cover, with examples, the main aspects of **pandas** which is an open source library providing high-performance, easy-to-use data structures and data analysis tools for Python.

10.1 Pandas & Dataframes

We already have covered the library **numpy** and how it is useful to store data in a rectangular structure. But there is a downside to this: you can only have data of the same type in there. In practice, you'll be working with data of different types: numerical values, strings, booleans and so on. To easily and efficiently handle this data, we use Python Data Analysis Library **pandas** which is a high-level data manipulation tool mainly used for data analysis. **pandas** allows importing data from various file formats such as comma-separated values, JSON, SQL, and Microsoft Excel. **pandas** allows various data manipulation operations such as merging, reshaping, selecting, as well as data cleaning. In **pandas**, we store data in a so-called dataframe. Dataframe is a two-dimensional labeled tabular data structure that contains rows which represent "observations or samples" and columns that represent "attributes or features". To clarify this, let's have a look at Fig(46) where it shows a built-in dataframe called "iris". This dataset consists of 3 different types of irises' (Setosa, Versicolour, and Virginica) petal and sepal lengths and widths, stored in a $m \times n = 150 \times 4$ table, where m and n denote the number of rows (samples) and columns (features), respectively. Each row has a unique row label: $0, 1, 2, ...$ that refers to a certain entry (or sample or observation) and each column is identified by its column label: Sepal Length, Sepal Width, Petal Length and Petal Width, where each one represents a feature (or an attribute) of this dataset.

10.2 Importing Datasets

The first step in any data analysis endeavour is to get the raw data into your program. Here, we will look at methods of loading data from two sources: a CSV

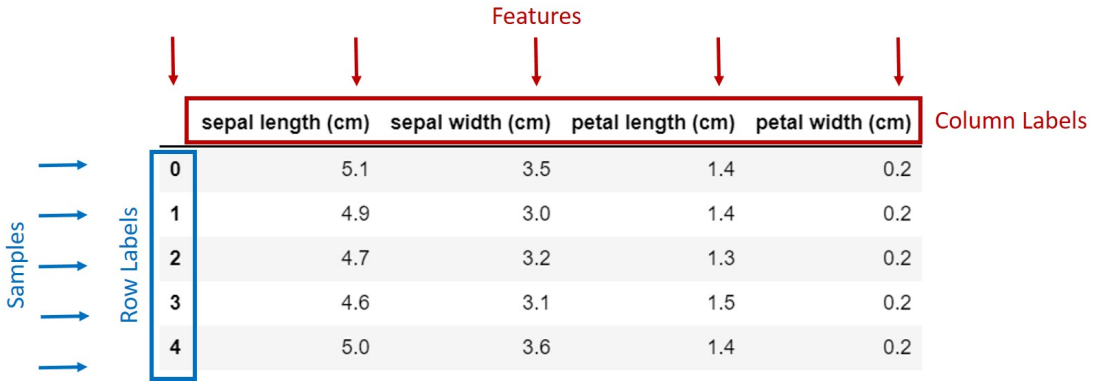


Figure 46: Dataframe Structure. Example: Iris Dataframe.

and Excel file.

10.2.1 Loading a CSV file

Comma-separated values (CSV) files with `.csv` extension represent plain text files that contain records of data with comma separated values. The CSV format is primarily used for storing tabular data, i.e., data that can be decomposed into rows and columns. Each row in a CSV file is a new record from the set of records contained in the file that, in turn, can be made up of one or more fields. Files with CSV format have several advantages [8]:

- CSV is easy to create. In fact, it can be done using any text editor.
- CSV format is human-readable, i.e., the data is not encoded or converted to binary before storing. This increases the readability as well as the ease of manipulation. Moreover, it is easy to edit.
- CSV files can be read using almost any text editor.

In Python, importing a comma-separated values (CSV) file has a specific syntax to follow. As shown in Fig(47), the code should start first by importing **pandas** library. Next, to import and read the CSV file, `read` method should be called from **pandas** library. In the Figure, a dataset movies.csv is imported and read by Python. You can surely take a glance of how this dataset is structures by calling the `head()` method that shows you the first five rows of the dataset, or you can call the `shape` method that shows you how many rows (samples) and columns (features) the dataset has. In this particular example, the dataset has 27278 movies and 3 features: movieID, title, and genres.

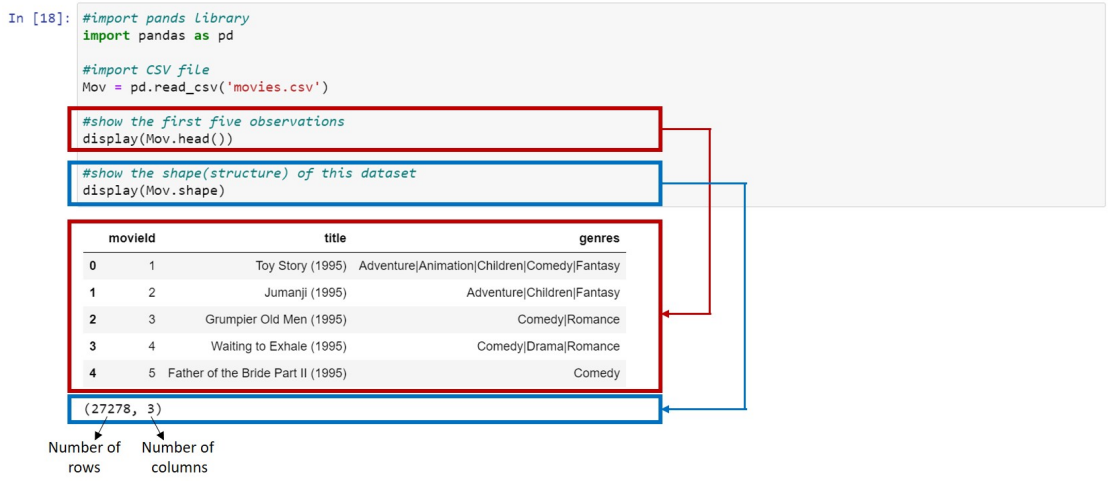


Figure 47: Python Syntax for importing a CSV file.

10.2.2 Loading an Excel file

Another type of formats that can be used to store tabular structure of data is Excel files. Excel files with `.xlsx` extension allow people to handle and store data in a controlled, easy, versatile, low cost, and simple manner. Excel's popularity and continuous expansion is due to the fact that Excel handles data in a flexible way and is compatible with several data file formats, i.e., it exports data to `.txt`, `.csv`, or `.xls` formats.

An excel file can be loaded in a similar way to a csv file. To import and read the excel file, `read` method should be called from **pandas** library. As shown in Fig(48), a dataset of 19 blood chemical properties (number of columns) taken from 30 participants in their rest condition (number of rows) is loaded and read by Python.

10.3 Understanding & Manipulating Data

Quite possibly the most important part in data analysis is understanding the data you are working with and how it relates to the task you want to solve. It will not be effective to randomly choose an algorithm and throw your data at it. It is necessary to understand what is going on in your dataset before you begin to analyse it, e.g., inspect for any abnormalities and peculiarities. In the following, we will cover a couple of main tools that are commonly used to clean and pre-process datasets.

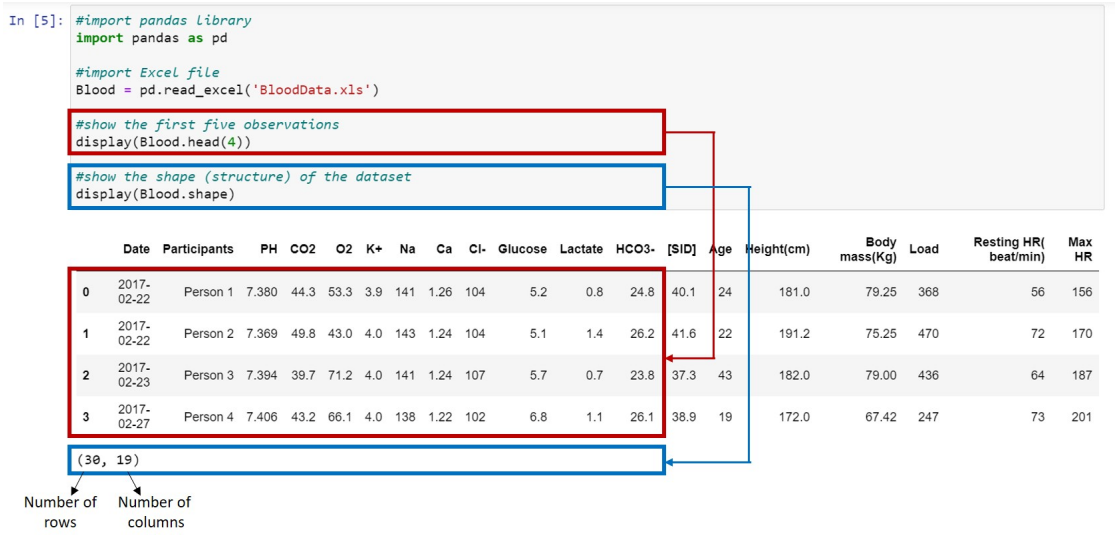


Figure 48: Python Syntax for importing an Excel file.

10.3.1 Data Inspection

After we load the dataset, it is a good idea to understand how it is structured and what kind of information it contains. Ideally, we would view the full data directly. But with most real-world cases, the data could have thousands to hundreds of thousands to millions of rows and columns. Instead, we have to rely on pulling samples to view small slices and calculating summary statistics of the data [5]. In Python, there are a couple of methods that are usually used to initially inspect the structure and the information of the dataset being analysed. We will go through them using the `Blood` dataset mentioned before as a toy example. First and foremost, it's very helpful to know the dimensionality (or the structure) of your dataset, i.e., how many rows (observations) and how many columns (features). This, as shown in Fig(49), can be done using the `shape` method. It displays two numbers, the first represents the number of rows, and the second represents the number of columns.

After knowing how many rows and columns there are in the dataset, it is quite helpful to have a look at the dataset itself. There are two methods to use in this case: `head()` and `tail()`. As shown in Fig(50), the former prints the first five observations and the latter prints the last five observations. Worth noting here that the default number of elements to display is five, but you may pass a custom number.

Another useful method to mention is `describe()`. This method is used to compute and view some basic statistical details like percentile, mean, median, standard deviation, minimum, maximum, etc. of a dataframe or a series of numeric

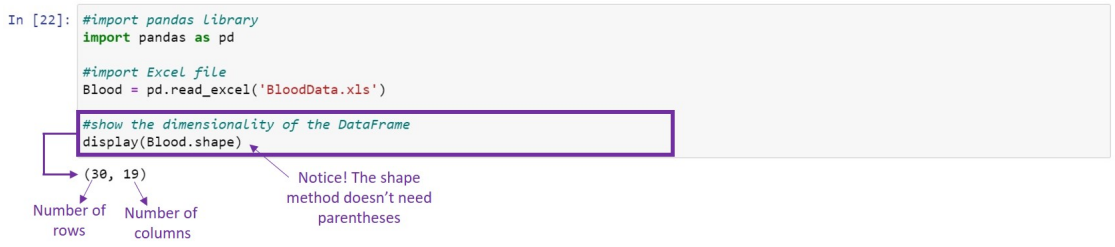
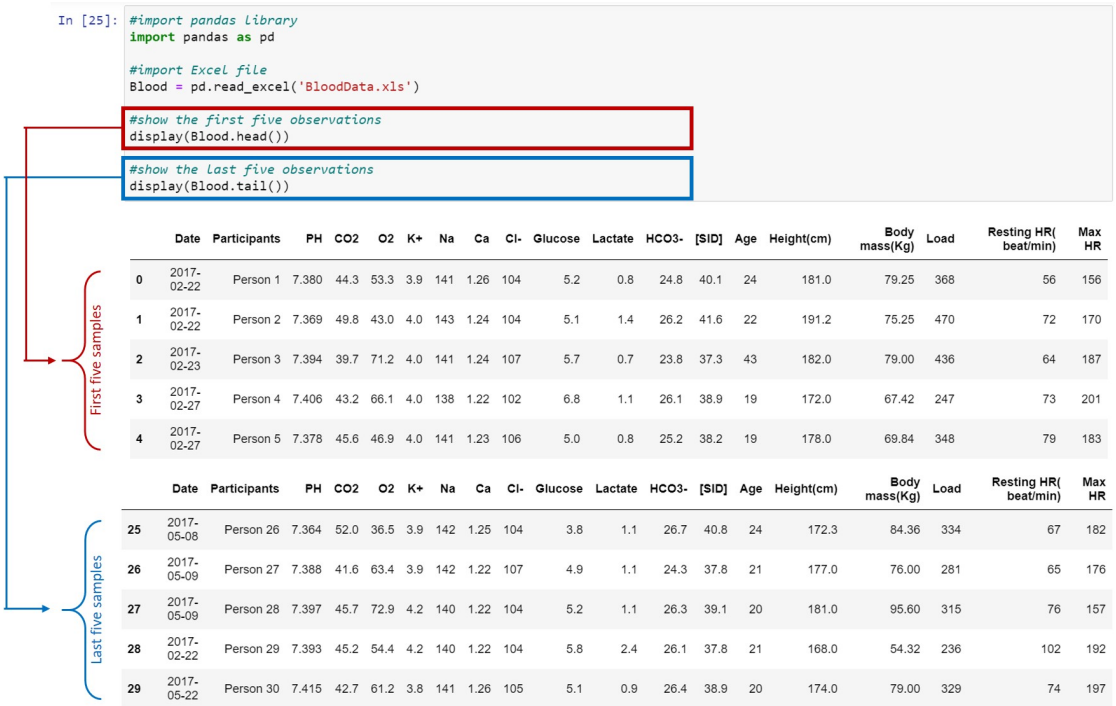


Figure 49: The `shape` method to print the structure of the dataframe.



	Date	Participants	PH	CO2	O2	K+	Na	Ca	Cl-	Glucose	Lactate	HCO3-	[SID]	Age	Height(cm)	Body mass(Kg)	Load	Resting HR(beat/min)	Max HR
0	2017-02-22	Person 1	7.380	44.3	53.3	3.9	141	1.26	104	5.2	0.8	24.8	40.1	24	181.0	79.25	368	56	156
1	2017-02-22	Person 2	7.369	49.8	43.0	4.0	143	1.24	104	5.1	1.4	26.2	41.6	22	191.2	75.25	470	72	170
2	2017-02-23	Person 3	7.394	39.7	71.2	4.0	141	1.24	107	5.7	0.7	23.8	37.3	43	182.0	79.00	436	64	187
3	2017-02-27	Person 4	7.406	43.2	66.1	4.0	138	1.22	102	6.8	1.1	26.1	38.9	19	172.0	67.42	247	73	201
4	2017-02-27	Person 5	7.378	45.6	46.9	4.0	141	1.23	106	5.0	0.8	25.2	38.2	19	178.0	69.84	348	79	183

	Date	Participants	PH	CO2	O2	K+	Na	Ca	Cl-	Glucose	Lactate	HCO3-	[SID]	Age	Height(cm)	Body mass(Kg)	Load	Resting HR(beat/min)	Max HR
25	2017-05-08	Person 26	7.364	52.0	36.5	3.9	142	1.25	104	3.8	1.1	26.7	40.8	24	172.3	84.36	334	67	182
26	2017-05-09	Person 27	7.388	41.6	63.4	3.9	142	1.22	107	4.9	1.1	24.3	37.8	21	177.0	76.00	281	65	176
27	2017-05-09	Person 28	7.397	45.7	72.9	4.2	140	1.22	104	5.2	1.1	26.3	39.1	20	181.0	95.60	315	76	157
28	2017-02-22	Person 29	7.393	45.2	54.4	4.2	140	1.22	104	5.8	2.4	26.1	37.8	21	168.0	54.32	236	102	192
29	2017-05-22	Person 30	7.415	42.7	61.2	3.8	141	1.26	105	5.1	0.9	26.4	38.9	20	174.0	79.00	329	74	197

Figure 50: To display the first or last five rows of a dataframe, two methods can be applied: `head()` and `tail()`, respectively.

values, as shown in Fig(51).

Last method to mention is `info()`. This method allows you to have a quick overview of your dataframe contents. It comes really handy when doing exploratory analysis of the data. As shown in Fig(52), it is used to print a concise summary of the dataframe including column data types, non-null values and memory usage.

```
In [23]: #import pandas library
         import pandas as pd

         #import Excel file
         Blood = pd.read_excel('BloodData.xls')

         #Generate descriptive statistics
         display(Blood.describe())
```

	PH	CO2	O2	K+	Na	Ca	Cl-	Glucose	Lactate	HCO3-	[SID]	Age	Height(cm)	mas
count	30.000000	30.000000	30.000000	30.000000	30.000000	30.000000	30.000000	30.000000	30.000000	30.000000	30.000000	30.000000	30.000000	30.0
mean	7.390433	44.930000	55.473333	3.973333	141.466667	1.225667	104.533333	5.223333	1.106667	25.676667	39.800000	24.066667	178.293333	76.2
std	0.023322	4.135561	14.467799	0.204995	1.357821	0.030021	1.382984	0.681116	0.419304	1.074367	1.909865	6.781991	5.478795	8.5
min	7.341000	36.300000	31.300000	3.600000	138.000000	1.170000	102.000000	3.400000	0.400000	22.900000	35.900000	19.000000	168.000000	54.3
25%	7.373750	42.725000	47.000000	3.825000	141.000000	1.212500	104.000000	5.000000	0.825000	25.125000	38.550000	20.000000	175.700000	70.8
50%	7.391000	44.650000	52.800000	3.950000	142.000000	1.230000	104.000000	5.200000	1.100000	25.750000	39.550000	21.000000	178.000000	76.0
75%	7.403750	47.975000	69.250000	4.100000	142.750000	1.247500	105.000000	5.625000	1.275000	26.200000	41.400000	24.000000	181.750000	80.0
max	7.442000	52.100000	80.100000	4.500000	143.000000	1.290000	107.000000	6.800000	2.400000	29.000000	43.600000	43.000000	191.200000	95.6

Figure 51: The `describe()` method is used to show the main statistical information of the dataframe.

```
In [24]: #import pandas library
         import pandas as pd

         #import Excel file
         Blood = pd.read_excel('BloodData.xls')

         #Print a concise summary of a DataFrame
         display(Blood.info())
```

```
<class 'pandas.core.frame.DataFrame'>          ⟶ Type of dataset
RangeIndex: 30 entries, 0 to 29                ⟶ Structure of DataFrame
Data columns (total 19 columns):               ⟶ Columns Information
 #   Column          Non-Null Count  Dtype     ⟶ Data type
---  ------          --------------  -----
 0   Date            30 non-null     datetime64[ns]
 1   Participants    30 non-null     object
 2   PH              30 non-null     float64
 3   CO2             30 non-null     float64
 4   O2              30 non-null     float64
 5   K+              30 non-null     float64
 6   Na              30 non-null     int64
 7   Ca              30 non-null     float64
 8   Cl-             30 non-null     int64
 9   Glucose         30 non-null     float64
 10  Lactate         30 non-null     float64
 11  HCO3-           30 non-null     float64
 12  [SID]           30 non-null     float64
 13  Age             30 non-null     int64
 14  Height(cm)      30 non-null     float64
 15  Body mass(Kg)   30 non-null     float64
 16  Load            30 non-null     int64
 17  Resting HR( beat/min)  30 non-null  int64
 18  Max HR          30 non-null     int64
dtypes: datetime64[ns](1), float64(11), int64(6), object(1)
memory usage: 4.6+ KB                          ⟶ Memory Usage

None
```

Missing values

Figure 52: The `info()` method is used to have a quick overview of the dataframe.

10.3.2 Slicing, Selecting, & Deleting Data

In data analysis, there is almost always a need to slice a dataset (chop the entire dataset into a sub-dataset), select individual rows and/or columns, or even delete rows and/or columns to get the right selection of data for the specific analysis in hand. In the following, we will cover the main syntaxes for slicing, selecting and

deleting rows and columns taking the `Blood` dataset as a toy example.

10.3.2.1 Slicing All rows and columns in a **pandas** dataframe have a unique index value. By default, this index is an integer indicating the row/column position in the dataframe; however, it does not have to be. Dataframe indices can be set to be unique alphanumeric strings or customer numbers. To select individual rows/columns or slices of rows/columns, **pandas** provides two methods [8]:

- `loc`: is useful when the index of the dataframe is a label (e.g., a string).
- `iloc`: works by looking for the position in the dataframe. For example, `iloc[0]` will return the first row regardless of whether the index is an integer or a label.

As shown in Fig(53), to slice out a set of rows, we use either of the following syntaxes:

```
data.iloc[start row index:stop row index,:]
data.loc[start row label:stop row label,:]
```

When slicing in pandas the start bound is included in the output. The stop bound is one step beyond the row you want to select. The colon here brings all the columns in the sliced subset. On the other hand, to slice out a set of columns, we use similar syntaxes as before:

```
data.iloc[:,start column index:stop column index]
data.loc[:,start column label:stop column label]
```

Similarly here, the stop bound is one step beyond the column you want to select. The colon here brings all the rows in the sliced subset. Sometimes, you might be interested to slice out both rows and columns of a given dataset. This can be done using one of the following syntaxes:

```
data.iloc[start row index:stop row index,start column index:stop column index]
data.loc[start row label:stop row label,start column label:stop column label]
```

```
In [33]: #import pandas library
         import pandas as pd

         #import Excel file
         Blood = pd.read_excel('BloodData.xls')

         #Slicing Subsets of Rows
         Blood.iloc[0:3,:] # Select rows 0, 1, 2 (row 3 is not selected) by index

         #Slicing Subsets of Columns
         Blood.iloc[:,4:7] # Select columns 4,5,6 (column 7 is not selected) by index
         #Blood.loc[:,"O2":"Na"] #by label

         #Slicing Subsets of Rows and Columns
         Blood.iloc[0:3,4:7] #Select rows 0, 1, 2 (row 3 is not selected) and columns 4,5,6 (column 7 is not selected) by index
         #Blood.loc[0:3,"O2":"Na"] #by label
```

Figure 53: Slicing subsets of rows,columns, and both.

10.3.2.2 Selecting Instead of slicing out the dataset, vertically or horizontally, you might be interested in only certain rows or columns. In this case, as shown in Fig(54), to select a particular number of rows and columns, the same methods of `iloc` and `loc` apply here by specifying their indices or the labels, respectively. Also, to select a single value from the dataset, the following syntax is used: `data.iloc[a,b]` where `a` is the index of the row and `b` is the index of the column where the element lie in.

```
In [44]: #import pandas library
         import pandas as pd

         #import Excel file
         Blood = pd.read_excel('BloodData.xls')

         #select one row
         Blood.iloc[5,:] #by index

         #select multiple rows
         Blood.iloc[[5,7],:] #by index

         #select one column
         Blood.iloc[:,3] #by index
         #Blood.loc[:,"CO2"] #by label

         #select multiple columns
         Blood.iloc[:,[3,8]] #by index
         #Blood.loc[:,["CO2","O2"]] #by label

         #Select list of rows and columns.
         Blood.iloc[[1,6,8],[0,6,9]] #by index

         #extract the data at the specified row and column location.
         Blood.iloc[0,3] #show the value resides in the row with index 0 and column with index 3
```

Figure 54: Selecting rows, columns, and both.

Sometimes, you need to access the last row (or column) in your dataframe. Interestingly, this can be done in Python using the `[-1]` index. Fig(55) shows how to extract the last row, last column, and the last element from last row and last column.

10.3.2.3 Deleting Also, sometimes you are interested in deleting one or more columns or rows out of the dataset you have. This can be easily done using the `drop` method as shown in Fig(56). To delete one or multiple rows, the process

```
In [4]: #import pandas library
        import pandas as pd

        #import Excel file
        Blood = pd.read_excel('BloodData.xls')

        #Select last row
        Blood.iloc[[-1],:]

        #Select last column
        Blood.iloc[:,[-1]]

        #Select last element from the last row and last column
        Blood.iloc[[-1],[-1]]
```

Figure 55: Selecting the last row, last column, and the last element from the last row and last column.

is easily done by identifying the index(ies) of the row(s) being deleted. However, to delete one or more columns the process is slightly different. The default way to use `drop` to remove columns is to provide the column indices (or labels) to be deleted along with specifying the `axis` parameter to be `1`.

```
In [58]: #import pandas library
         import pandas as pd

         #import Excel file
         Blood = pd.read_excel('BloodData.xls')

         #delete one row
         Blood.drop([0]) #by index

         #delete multiple rows
         Blood.drop([0,6]) #by index

         #delete one column
         Blood.drop(Blood.columns[0], axis=1) #delete the first column by index
         #Blood.drop('Participants', axis=1) #by label

         #delete multiple columns
         Blood.drop(Blood.columns[[0,1,3]], axis=1) #delete the first, second and fourth columns by index
         #Blood.drop(['PH','O2'], axis=1) #by label
```

Figure 56: Deleting rows and columns from a dataset.

10.3.3 Handling Missing Values

Real-world data would certainly have missing values. This could be due to many reasons such as data entry errors or data collection problems. Irrespective of the reasons, it is important to handle missing data because any statistical results based on a dataset with non-random missing values could be biased. Oftentimes, a dataset either shows an empty cell or uses a specific value to denote a missing observation, such as `NONE`, `-999`, `n/a`, `NA`, or `NaN`. To transform the dataset into a clean and organised format ready to be used for a data analysis tasks, these missing values should be dealt with properly. There is no generic way to follow when it comes to missing values. The choice of how to deal with them depends entirely with the nature of the dataset being analysed. Having said that, there

are some common practices which are usually used to tackle such values. For example, dropping columns with missing values, filling in the missing values with some number, such as zero, or the mean of the row or column the value lies in. In Python, you can identify whether a certain column has any missing values by using the `isnull()` method, as shown in Fig(57). The output is a boolean parameter that indicates whether there is (`True`) or isn't (`False`) any missing values in each row of that column.

```
In [67]: #import pandas library
         import pandas as pd

         #import Excel file
         Blood = pd.read_excel('BloodData.xls')

         #Check for any missing values in a specific column
         Blood['Participants'].isnull()

Out[67]: 0     False
         1     False
         2     False
         3     False
         4     False
         5     False
         6     False
         7     False
         8     False
         9     False
         10    False
         11    False
         12    False
         13    False
         14    False
         15    False
         16    False
         17    False
         18    False
         19    False
         20    False
         21    False
```

Figure 57: `isnull()` method to check whether the column "Participants" has any missing values.

Another useful way is to see that total number of missing values for each feature in the dataset. This can be done by two methods `isnull()` and `sum()`, as shown in Fig(58).

```
In [68]: #import pandas library
         import pandas as pd

         #import Excel file
         Blood = pd.read_excel('BloodData.xls')

         #Check for any missing values in a specific column
         Blood['Participants'].isnull()

         #Show the total missing values for each feature
         Blood.isnull().sum()

         #drop all rows that have missing data
         #Blood.dropna(how="all", inplace=True)
```

Figure 58: `isnull()` and `sum()` methods are used to show the total number of missing values in each column.

11 Data Visualisation

Data visualization is the graphical representation of data or information by using elements like graphs, charts, or other visual format. It communicates relationships of the data with images. This is important because it allows trends and patterns to be more easily seen. We need data visualization because a visual summary of information makes it easier to identify patterns and trends than looking through thousands of rows on a spreadsheet. It's the way the human brain works. Since the purpose of data analysis is to gain insights, data is much more valuable when it is visualized. There are numerous tools available to help create data visualizations. Python offers multiple great graphing libraries that come packed with lots of different features. In this section, we will learn how to create basic plots using **matplotlib** library and how to use some of its specific features.

11.1 Figures and Subplots

Generally, to create a visual output, the very first step is to import **matplotlib** library before using any of its methods. In Python, all plotting functions exist inside a sub-module of **matplotlib** called **pyplot**. **pyplot** is an interface that contains a collection of command-style functions that we can use for performing common actions on our plot. Therefore, this sub-module needs to be imported as well so its functions can be called later on in the code. Here, we will be using the `Blood` dataset as a toy example for illustrating the different built-in figure functions included in this library.

11.1.1 Scatter Plot

Scatter plots graph pairs of numerical data, one variable on each axis, to look for a relationship using different points called 'markers', e.g., dots, circles, crosses, etc. The variable on the x-axis is called the independent variable, whereas the variable on the y-axis is called the dependent variable. Scatter plots shows the degree and kind of correlation between the both variables. This can be one of the cases shown in Fig(59).

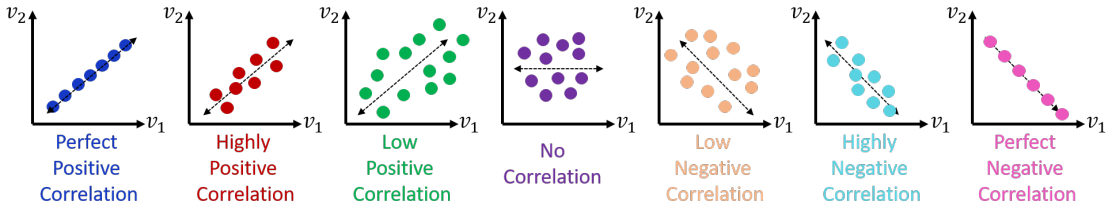


Figure 59: Different correlation patterns that can be visualised by scatter plots.

To create a scatter plot in **matplotlib** we can use the `scatter` method. As shown in Fig(60), four main steps are followed to perform such task:

- Import relevant libraries. Two libraries here are imported: **pandas** for reading the dataset and **matplotlib** for data visualisation.
- Specify data to plot. After reading the dataset, we need to choose what data to plot. Here, we select two columns: CO2 on the x axis and O2 y axis.
- Call the `scatter` method to generate the plot.
- Show the plot by calling the `show()` method.

```
In [7]: #import pandas library
        import pandas as pd
        #import matplotlib library
        import matplotlib.pyplot as plt

        #import and read the dataset
        Blood = pd.read_excel('BloodData.xls')
        #Specify data to plot
        x = Blood.loc[:, 'CO2'] #Define the x axis data
        y = Blood.loc[:, 'O2'] #Define the y axis data

        #call scatter method to plot x and y
        plt.scatter(x, y)

        #show the plot
        plt.show()
```

Import libraries

Specify data to plot

Plot the data

Show the plot

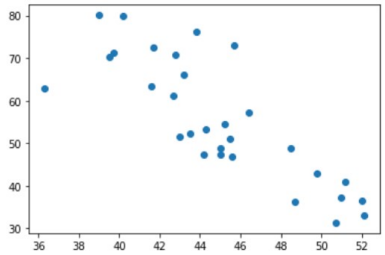


Figure 60: The main steps to generate a scatter plot in Python.

11.1.2 Line Chart

A line chart or line plot or line graph is a type of chart which displays the information of a data set as a series of data points connected by a straight line. In **matplotlib** we can create a line chart by calling the `plot` method. Although similar main steps to scatter plots are followed here to generate a line chart, as shown in Fig(61), two differences can be noticed:

1. Only the CO2 column is specified to plot. The numbers of this column are interpreted as the y-values to create the plot. The x axis, on the other hand, is scaled automatically by the **pyplot** interface. Otherwise, you can surely specify certain values or a range for x axis.

2. The `plot` function is called.

```
In [13]: #import pandas library
         import pandas as pd
         #import matplotlib library
         import matplotlib.pyplot as plt

         #import and read the dataset
         Blood = pd.read_excel('BloodData.xls')
         #Specify data to plot
         x = Blood.loc[:, 'CO2'] #Define the data to plot

         #call line method to plot x
         plt.plot(x)

         #show the plot
         plt.show()
```

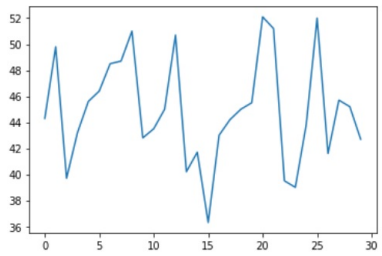
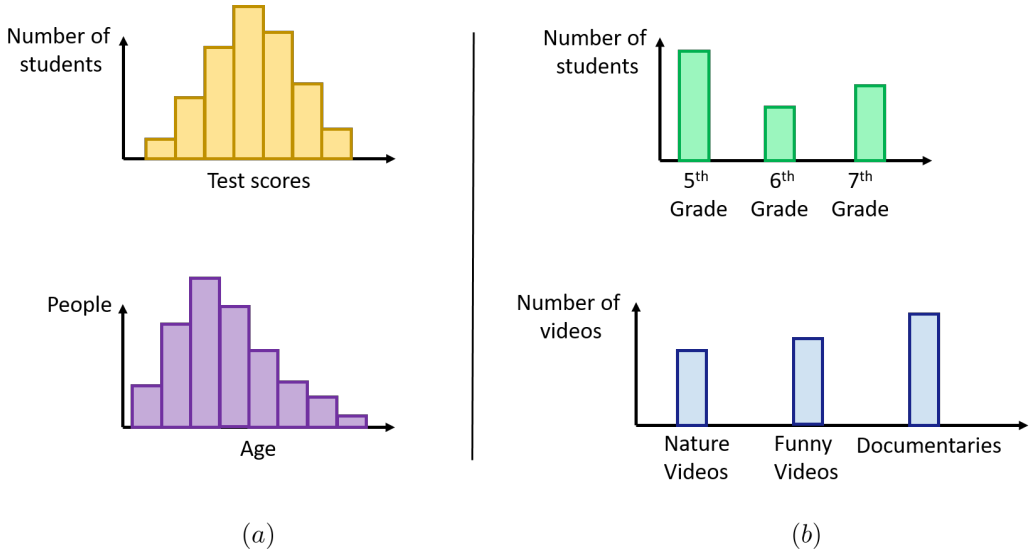


Figure 61: The main steps to generate a line chart in Python.

11.1.3 Histogram

Histograms are the most commonly used graph for showing frequency distributions, or how often each different value in a set of data occurs. This kind of graph uses vertical bars to display quantitative data. The heights of the bars indicate the frequencies of values in our data set. It is very important to mention here that Histograms should only be used for continuous data. However, for discrete (categorical) data we should use bar charts. As shown in Fig(62,a), in a histogram,

the categories are numerical (continuous, quantities) data. Bars usually are touching. However, in a bar graph in Fig(62,b), the categories are usually categorical (discrete) data. Bars usually are separated. In **matplotlib** we can create a Histogram using the `hist` method, as shown in Fig(63), following similar main steps as before.



(a) (b)

Figure 62: Difference between histograms in (a) and bar charts in (b).

```
In [14]: #import pandas library
         import pandas as pd
         #import matplotlib library
         import matplotlib.pyplot as plt

         #import and read the dataset
         Blood = pd.read_excel('BloodData.xls')
         #Specify data to plot
         x = Blood.loc[:, 'CO2']

         #call hist method
         plt.hist(x)

         #show the histogram
         plt.show()
```

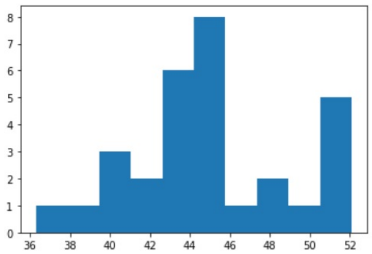


Figure 63: Histograms syntax in Python.

11.1.4 Bar Chart

As mentioned before, a bar chart or bar graph is a chart or graph that presents categorical data with rectangular bars with heights or lengths proportional to the values that they represent. The bars can be plotted vertically or horizontally. In Python, a bar chart can be created using the `bar` method. Fig(64) shows three bars that represent the CO2 values for the first three participants. You can notice that the quantities on the x-axis are categorical in this case rather than numerical as before.

```
In [42]: #import pandas library
         import pandas as pd
         #import matplotlib library
         import matplotlib.pyplot as plt

         #import and read the dataset
         Blood = pd.read_excel('BloodData.xls')
         #Set the bar labels
         participants = ['Participant 1','Participant 2', 'Participants 3']
         #select the values for each bar
         CO2 = [Blood['CO2'].iloc[0], Blood['CO2'].iloc[1], Blood['CO2'].iloc[2]]

         #plot the bar chart
         plt.bar(participants,CO2)

         #show the chart
         plt.show()
```

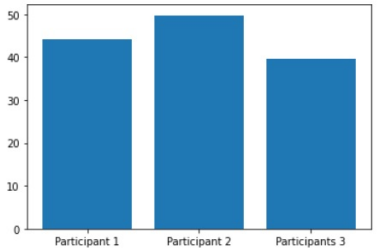


Figure 64: Bar chart syntax in Python.

11.1.5 Subplots

To draw multiple plots in one figure we use the `subplot()` function. The `subplot()` function takes three arguments that describes the layout of the figure. The layout is organized in rows and columns, which are represented by the first and second argument. The third argument represents the index of the current plot. For example, if we want two plots to be vertically stacked with 2 rows and 1 column (meaning that the two plots will be displayed on top of each), we can write the syntax as shown in Fig(65,a). However, if we want the two plots to be horizontally stacked with 1 row and 2 columns (meaning that the two plots will be displayed side by side), we can write the syntax as shown in Fig(65,b). Generally speaking, we can plot as many subplots as the problem requires. For example, as shown in Fig(65,c), we can organise a 2×2 layout for four plots in the same figure.

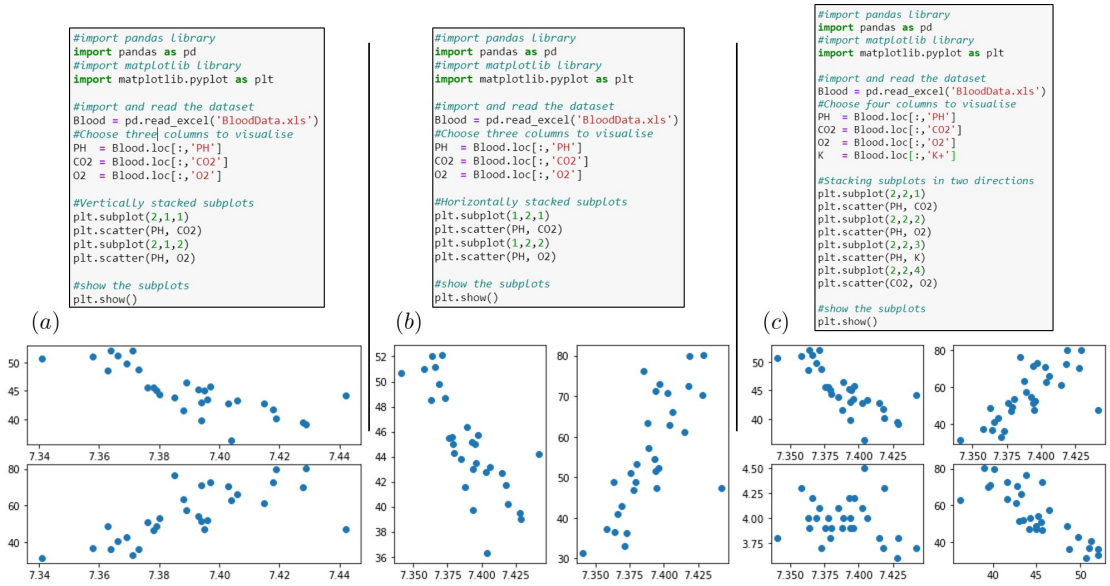


Figure 65: (a) Two subplots stacked vertically. (b) Two subplots stacked horizontally. (c) Four subplots stacked in a 2D structure.

11.2 Title, Axes Labels & Legends

It is essential to keep in mind that any plot that visualises a certain set of data needs to have the following main parts:

1. Figure.
2. Title.
3. Axes labels
4. Legend

Missing any of the above parts results a major loss of information and misinterpreting the ideas the plot tries to show. Since we have covered how to plot several types of figures in Python earlier, we will cover here how to include the rest of the main parts. For simplicity, as the steps apply to all other types of plots, we will focus on the scatter plot for illustration. Fig(66) shows two scatter plots: the first plotting PH as a function of CO2 in blue dots and the second is plotting PH as a function of O2 in orange dots. After calling the `scatter` method to plot both data sets, a title, axis labels and legend are specified to be added to the plot. It is always important to remember adding these information as they certainly show the value of what the plot displays.

```python
#import pandas library
import pandas as pd
#import matplotlib library
import matplotlib.pyplot as plt

#import and read the dataset
Blood = pd.read_excel('BloodData.xls')
#Specify data to plot
PH  = Blood.loc[:,'PH']
CO2 = Blood.loc[:, 'CO2']
O2  = Blood.loc[:, 'O2']

#call scatter method
plt.scatter(PH,CO2) #scatter PH as a function of CO2
plt.scatter(PH,O2) #scatter PH as a function of O2

#add title
plt.title('Blood Chemistry')

#add axis labels
plt.xlabel('CO2 and O2')
plt.ylabel('PH')

#add legend
plt.legend(['CO2','O2'])

#show the plot
plt.show()
```

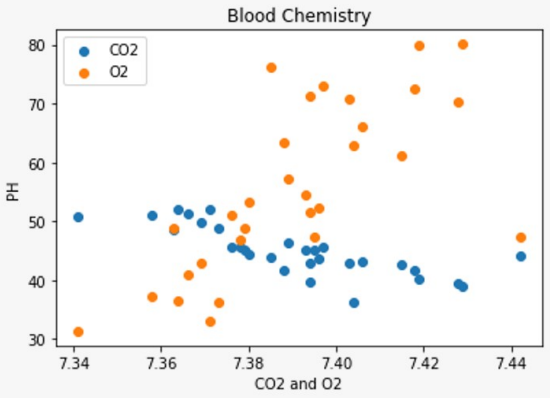


Figure 66: Title, axes labels and legend in Python.

11.3 Problem: Ideal Gas Behaviour

Quantities such as pressure, volume, temperature, and amount of substance describe the conditions ,or *state*, in which a particular material exists. These quantities are called state variables [6]. The volume V of a substance is usually determined by its pressure P, temperature T, and amount of substance, described by the mass m or number of moles n. Ordinarily, we can't change one of these variables without causing a change in another. Measurements of the behaviour of various gases lead to three conclusions (or laws)

1. The pressure is proportional to the absolute temperature T. If we double T, keeping the volume and number of moles constant, the pressure doubles. In other words, $P \propto T$ or $P = \text{constant} \times T$ when n and V are constant. This is usually referred as *Amontons's Law* or *Gay-Lussac's law*.

2. The volume is proportional to the absolute temperature T. The volume increases as the temperature increases, and decreases as the temperature decreases by keeping the volume and number of moles constant. In other words, $V \propto T$ or $V = \text{constant} \times T$ when n and P are constant.The relationship between the volume and temperature of a given amount of gas at constant pressure is known as *Charles's law*.

3. Unlike the P–T and V–T relationships, pressure and volume are not directly proportional to each other. Instead, P and V exhibit inverse proportionality: Increasing the pressure, with holding the temperatures T and number of

moles n constant, results in a decrease of the volume of the gas. In other words, $P \propto 1/V$ or $PV =$ constant when n and T are constant. This known as *Boyle's law*.

4. The volume V is proportional to the number of moles n. If we double n, keeping pressure and temperatures constant, the volume doubles. This relationship is usually referred as *Avogadro's law*.

When we combine these three relationships into a single ideal-gas equation:

$$PV = nRT \tag{5}$$

In SI units, the unit of P is Pa, and the unit of V is m^3. We might expect that the proportionality constant R in Eq(5) would have different values for different gases, but it turns out to have the same value for *all* gases. It is called the gas constant and its numerical value is equal to $R = 8.314$J/mol.K. An ideal gas is one for which Eq(5) holds precisely for *all* pressures and temperatures. This is an idealised model; it works best at very low pressures and high temperatures, when the gas molecules are far apart and in rapid motion. Fig(67) shows a Python code that plots the main relationships (or laws) of an ideal gas.

In [15]:

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

#Define gas constant
R = 8.314;

#P-T relationship: Amontons's Law or Gay-Lussac's law
Temp = np.linspace(273.15,50+323.15,100); #variable
V    = 0.0132; #constant
n    = 50; #constant
P_Amonton= n*R*Temp/V; #solve for P

#V-T relationship: Charles's law
Temp = np.linspace(273.15,50+323.15,100); #variable
P    = 15502725; #constant
n    = 50; #constant
V_Charles= n*R*Temp/P; #solve for V

#P-V relationship: Boyle's law
Volume = np.linspace(0.001,0.03,100); #variable
n      = 50; #constant
T      = 300.15; #constant
P_Boyle= n*R*T/Volume; #Solve for P

#V-n relationship: Avogadro's law
nmole = np.linspace(1,500,100); #variable
P     = 15502725; #constant
T     = 300.15; #constant
V_Avogadro= nmole*R*T/P; #Solve for V

#Stacking subplots in two directions
fig = plt.figure() #generate the figure
fig.subplots_adjust(hspace=0.9, wspace=0.6) #adjusting the spaces between the subplots
#P-T relationship: Amontons's Law or Gay-Lussac's law
plt.subplot(2,2,1)
plt.plot(Temp, P_Amonton, c='green')
plt.title('Amontons Law')
plt.xlabel('T $(K)$')
plt.ylabel('P $(Pa)$')
#V-T relationship: Charles's law
plt.subplot(2,2,2)
plt.plot(Temp, V_Charles, c='blue')
plt.title('Charles  law')
plt.xlabel('T $(K)$')
plt.ylabel('V $(m^3)$')
#P-V relationship: Boyle's law
plt.subplot(2,2,3)
plt.plot(Volume, P_Boyle, c='red')
plt.title('Boyle law')
plt.xlabel('V $(m^3)$')
plt.ylabel('P $(Pa)$')
#V-n relationship: Avogadro's law
plt.subplot(2,2,4)
plt.plot(nmole, V_Avogadro, c='orange')
plt.title('Avogadro law')
plt.xlabel('n $(mole)$')
plt.ylabel('V $(m^3)$')

#show the subplots
plt.show()
```

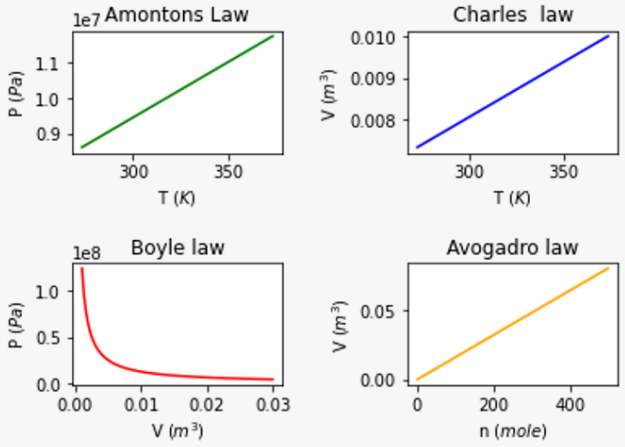


Figure 67: Python code to plot the four main laws that describe the basic relationships between the variables of state; pressure P, volume V, Temperature T, and number of moles n.

References

[1] Phillips D. *Python 3 Object-oriented Programming*. Packt Publishing, 2015.

[2] Giese T. Bulut S. Wende, M. and R. Anderl. Framework of an active learning python curriculum for first year mechanical engineering students. *IEEE Global Engineering Education Conference (EDUCON)*.

[3] Diaz C. Davim, J. P. and V. K. Solanki. *IEEE Global Engineering Education Conference (EDUCON)*. CRC Press, 2019.

[4] Mueller A. and Guido S. *Introduction to Machine Learning with Python: A Guide for Data Scientists*. O'Reilly Media, 2016.

[5] McKinney W. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, 2017.

[6] Young H. and Freedman R. *University Physics with Modern Physics: Fourteenth Edition*. Pearson Education Limited, 2016.

[7] Kutz N. *Data-Driven Modelling & Scientific Computation: Methods for Complex Systems & Big Data*. Oxford University Press, 2013.

[8] Albon C. *Machine Learning with Python Cookbook: Practical Solutions from Preprocessing to Deep Learning*. O'Reilly Media, 2018.