# COS30018 – Intelligent Systems

# Week 4 tutorial

This week we will cover the following items:

- Python introduction
- Google Colab introduction and walkthrough
- 1st machine Learning library: Scikit-learn (sklearn)
- 1st machine learning algorithm: Linear Regression

## 1. Python introduction

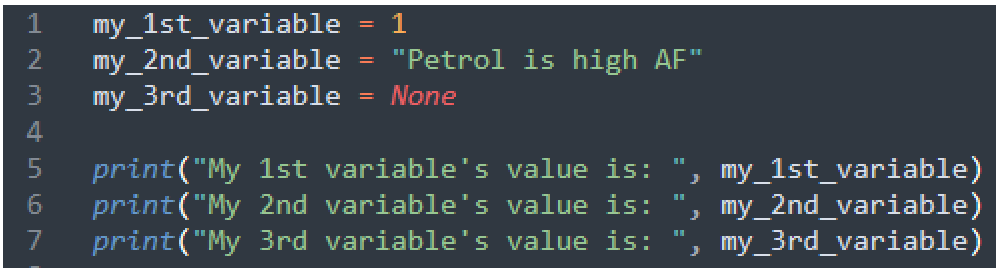
Benefits that make Python the best fit for machine learning and AI-based projects include simplicity and consistency, access to great libraries and frameworks for AI and machine learning (ML), flexibility, platform independence, and a wide community. These add to the overall popularity of the language.

Variables:

To declare a variable in Python, you just simply write the name you want to give to the variable and assign to it an initialised value. Different from Java, Python is a dynamically typed language, which means the types of the variables are dynamically determined during runtime. Hence, we do not need to specify the variable type, Python will figure it out when we assign a value to that variable (Your life is easier, isn't it?)

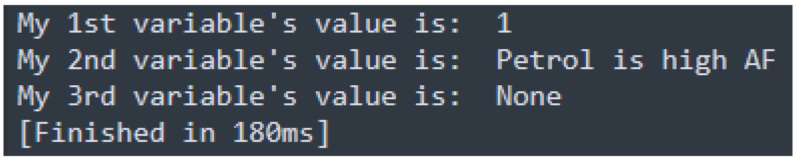
All variables in Python MUST be declared with an initialised value. However, you can assign it to "None" if you want that variable to be created without a value.

For example, you can declare variables like this, and print it out:



```python
my_1st_variable = 1
my_2nd_variable = "Petrol is high AF"
my_3rd_variable = None

print("My 1st variable's value is: ", my_1st_variable)
print("My 2nd variable's value is: ", my_2nd_variable)
print("My 3rd variable's value is: ", my_3rd_variable)
```

The output will be:



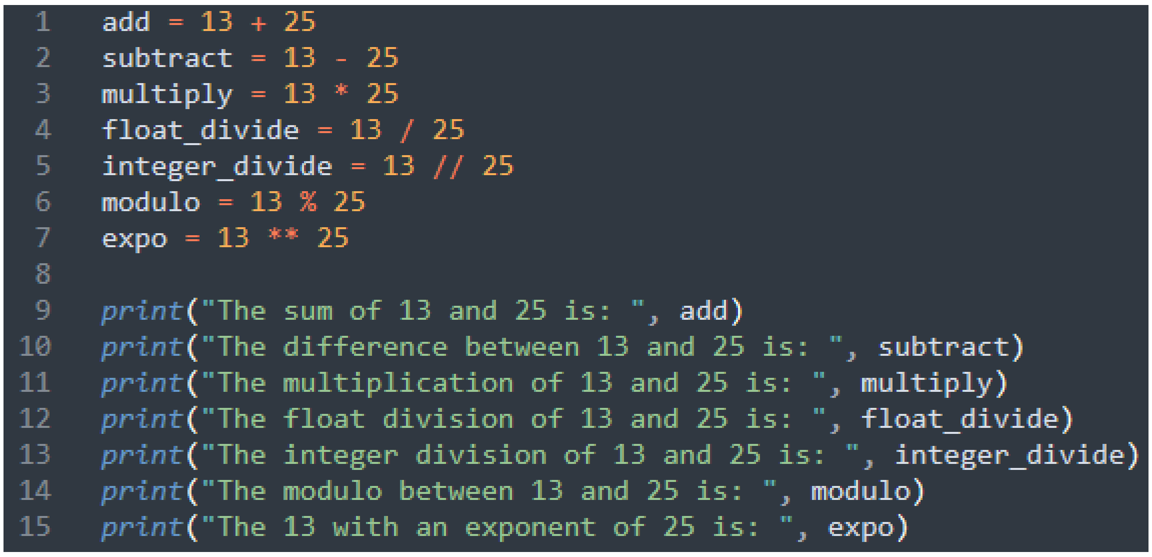
```
My 1st variable's value is:  1
My 2nd variable's value is:  Petrol is high AF
My 3rd variable's value is:  None
[Finished in 180ms]
```

Numbers, strings & operators:

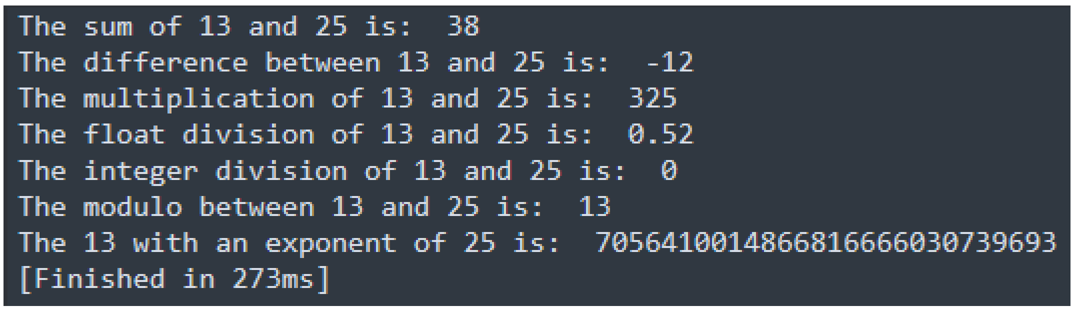
Just as any other programming languages, the addition, subtraction, multiplication, and division operators can be used with numbers. Another operator available is the modulo (%) operator, which returns the integer remainder of the division (dividend % divisor = remainder). You can use two multiplication symbols to do the exponent. The Arithmetic operators can be viewed in the below table:

| Operator | Name | Example |
|---|---|---|
| + | Addition | x + y |
| - | Subtraction | x - y |
| * | Multiplication | x * y |
| / | Division | x / y |
| % | Modulus | x % y |
| ** | Exponentiation | x ** y |
| // | Floor division | x // y |

Examples:

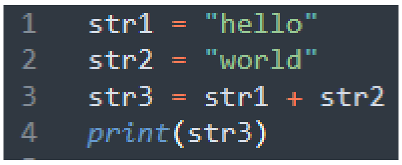


```python
add = 13 + 25
subtract = 13 - 25
multiply = 13 * 25
float_divide = 13 / 25
integer_divide = 13 // 25
modulo = 13 % 25
expo = 13 ** 25

print("The sum of 13 and 25 is: ", add)
print("The difference between 13 and 25 is: ", subtract)
print("The multiplication of 13 and 25 is: ", multiply)
print("The float division of 13 and 25 is: ", float_divide)
print("The integer division of 13 and 25 is: ", integer_divide)
print("The modulo between 13 and 25 is: ", modulo)
print("The 13 with an exponent of 25 is: ", expo)
```

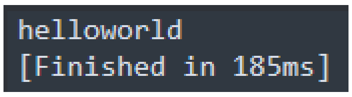


```
The sum of 13 and 25 is:  38
The difference between 13 and 25 is:  -12
The multiplication of 13 and 25 is:  325
The float division of 13 and 25 is:  0.52
The integer division of 13 and 25 is:  0
The modulo between 13 and 25 is:  13
The 13 with an exponent of 25 is:  7056410014866816666030739693
[Finished in 273ms]
```

For strings, Python supports concatenating strings using the addition operator:

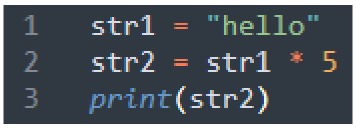


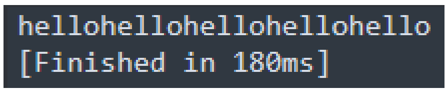
```python
str1 = "hello"
str2 = "world"
str3 = str1 + str2
print(str3)
```



```
helloworld
[Finished in 185ms]
```

Python also supports multiplying strings to form a string with a repeating sequence:


```python
str1 = "hello"
str2 = str1 * 5
print(str2)
```

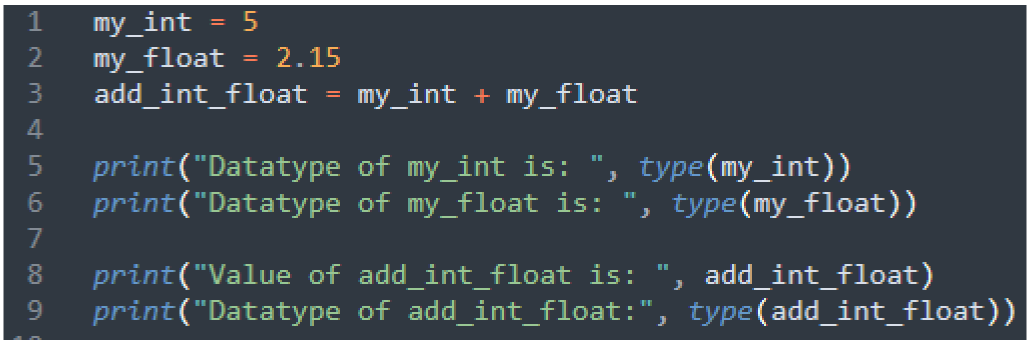

```
hellohellohellohellohello
[Finished in 180ms]
```

Type conversion:

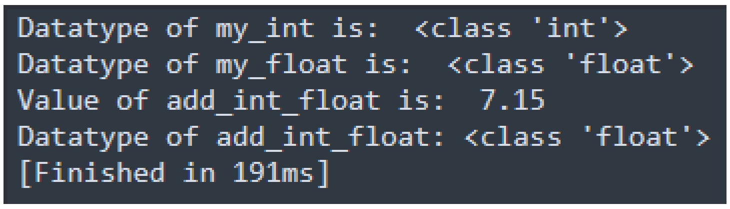
The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion. Python has two types of type conversion:

- Implicit Type Conversion
- Explicit Type Conversion

In Implicit type conversion, Python automatically converts one data type to another data type. This process doesn't need any user involvement. Let's see an example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss:

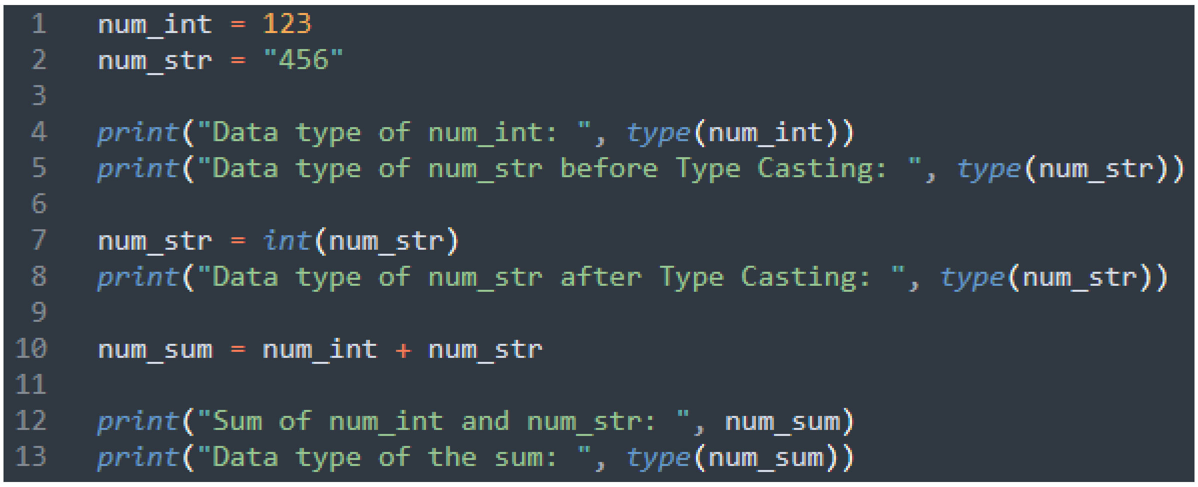

```python
my_int = 5
my_float = 2.15
add_int_float = my_int + my_float

print("Datatype of my_int is: ", type(my_int))
print("Datatype of my_float is: ", type(my_float))

print("Value of add_int_float is: ", add_int_float)
print("Datatype of add_int_float:", type(add_int_float))
```

The output of the above code is:


```
Datatype of my_int is:  <class 'int'>
Datatype of my_float is:  <class 'float'>
Value of add_int_float is:  7.15
Datatype of add_int_float: <class 'float'>
[Finished in 191ms]
```

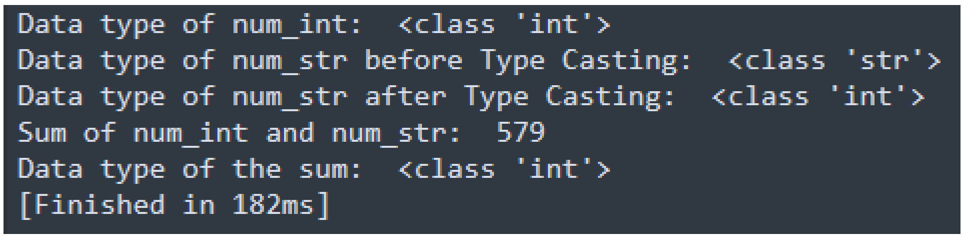
We can see that “my_int” is an integer number, but when it is added with a float number, Python internally converts “my_int” to float type, so that it can be added with “my_float”. Hence the result “add_int_float” is a float number.

In Explicit Type Conversion, users convert the data type of an object to required data type. We use the predefined functions like **int(), float(), str(),** etc to perform explicit type conversion. This type of conversion is also called **typecasting** because the user casts (changes) the data type of the objects. Let’s see an example below:



```python
num_int = 123
num_str = "456"

print("Data type of num_int: ", type(num_int))
print("Data type of num_str before Type Casting: ", type(num_str))

num_str = int(num_str)
print("Data type of num_str after Type Casting: ", type(num_str))

num_sum = num_int + num_str

print("Sum of num_int and num_str: ", num_sum)
print("Data type of the sum: ", type(num_sum))
```

The output will be:

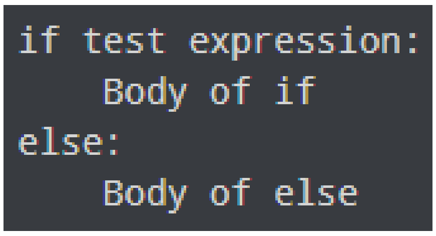


```
Data type of num_int:  <class 'int'>
Data type of num_str before Type Casting:  <class 'str'>
Data type of num_str after Type Casting:  <class 'int'>
Sum of num_int and num_str:  579
Data type of the sum:  <class 'int'>
[Finished in 182ms]
```

We converted “num_str” from string (higher) to integer (lower) type using int() function to perform the addition. After converting “num_str” to an integer value, Python is able to add these two variables.

If statements:

*Syntax of if...else:*



```
if test expression:
    Body of if
else:
    Body of else
```

The if...else statement evaluates test expression and will execute the body of if only when the test condition is True. If the condition is False, the body of else is executed. **Indentation** is used to separate the blocks. **Indentation** is extremely important in Python.

Example:

```
# Program checks if the number is positive or negative
# And displays an appropriate message

num = 3

# Try these two variations as well.
# num = -5
# num = 0

if num >= 0:
    print("Positive or Zero")
else:
    print("Negative number")
```

*Syntax of if...elif...else:*

```
if test expression:
    Body of if
elif test expression:
    Body of elif
else:
    Body of else
```

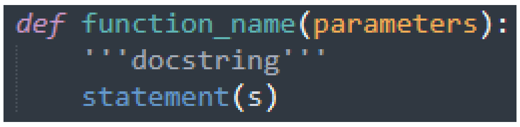
The elif is short for else if. It allows us to check for multiple expressions. If the condition for if is False, it checks the condition of the next elif block and so on. If all the conditions are False, the body of else is executed. Only one block among the several if...elif...else blocks is executed according to the condition.

Example:

```
'''In this program,
we check if the number is positive or
negative or zero and
display an appropriate message'''

num = 3.4

# Try these two variations as well:
# num = 0
# num = -4.5

if num > 0:
    print("Positive number")
elif num == 0:
    print("Zero")
else:
    print("Negative number")
```

## Function:

*Syntax of Function in Python:*



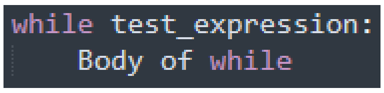
```
def function_name(parameters):
    '''docstring'''
    statement(s)
```

Above shown is a function definition that consists of the following components:

- Keyword `def` that marks the start of the function header.
- A function name to uniquely identify the function. Function naming follows the same rules of writing variables in Python.
- Parameters (arguments) through which we pass values to a function. They are optional.
- A colon (:) to mark the end of the function header.
- Optional documentation string (docstring) to describe what the function does.
- One or more valid python statements that make up the function body. Statements must have the same **indentation level** (usually 4 spaces).
- An optional `return` statement to return a value from the function.

## While loop:

*Syntax of* `While` *loop in Python:*



```
while test_expression:
    Body of while
```

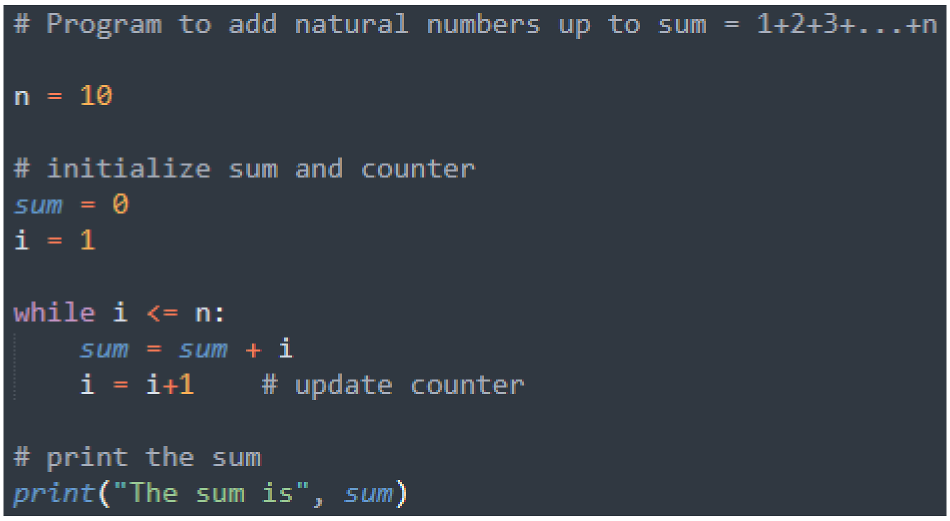
In the while loop, test expression is checked first. The body of the loop is entered only if the `test_expression` evaluates to `True`. After one iteration, the test expression is checked again. This process continues until the `test_expression` evaluates to `False`.

In Python, the body of the while loop is determined through **indentation** (told ya! Indentation in Python is very important).

The body starts with indentation and the first unindented line marks the end.

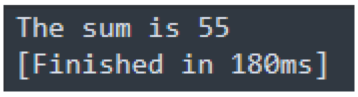
Python interprets any non-zero value as `True`. `None` and `0` are interpreted as `False`.

Example:



```
# Program to add natural numbers up to sum = 1+2+3+...+n

n = 10

# initialize sum and counter
sum = 0
i = 1

while i <= n:
    sum = sum + i
    i = i+1    # update counter

# print the sum
print("The sum is", sum)
```

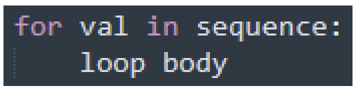
The output of the above code will be:



```
The sum is 55
[Finished in 180ms]
```

<u>For loop:</u>

*Syntax of `For` loop in Python:*

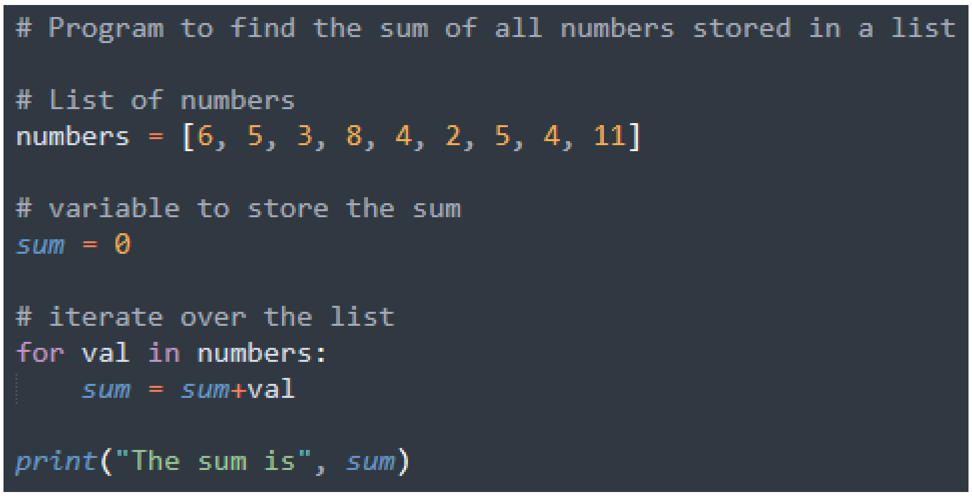


```
for val in sequence:
    loop body
```

Here, `val` is the variable that takes the value of the item inside the sequence on each iteration.

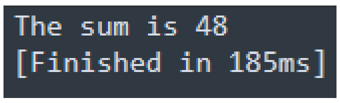
Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using **indentation**.

Example:



```python
# Program to find the sum of all numbers stored in a list

# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum
sum = 0

# iterate over the list
for val in numbers:
    sum = sum+val

print("The sum is", sum)
```

The output of the above code will be:

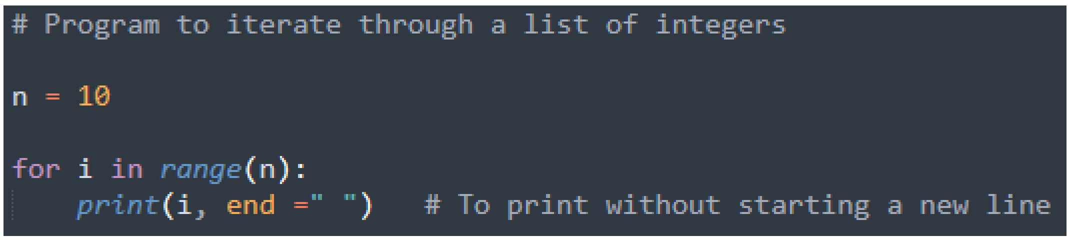


```
The sum is 48
[Finished in 185ms]
```

*The `range()` function:*

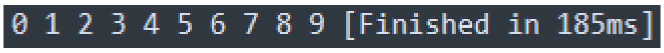
`range()` function is very commonly used with `for` loop. We can generate a sequence of numbers using `range()` function. `range(10)` will generate numbers from 0 to 9 (10 numbers). We can also define the start, stop and step size as `range(start, stop, step_size)`. `step_size` defaults to 1 if not provided.

Example 1:



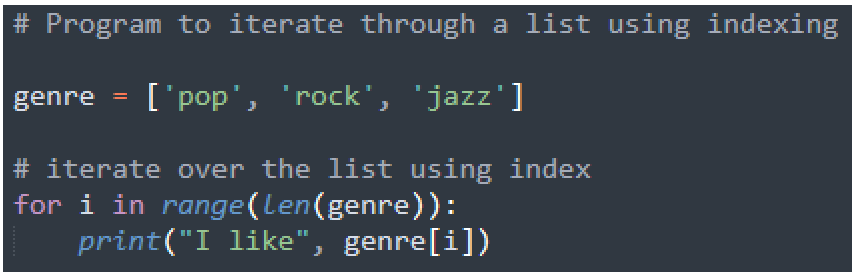
```python
# Program to iterate through a list of integers

n = 10

for i in range(n):
    print(i, end =" ")    # To print without starting a new line
```

Output will be:



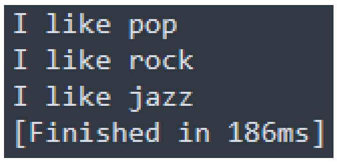
```
0 1 2 3 4 5 6 7 8 9 [Finished in 185ms]
```

Example 2:



```python
# Program to iterate through a list using indexing

genre = ['pop', 'rock', 'jazz']

# iterate over the list using index
for i in range(len(genre)):
    print("I like", genre[i])
```

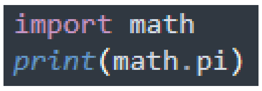
Output will be:



```
I like pop
I like rock
I like jazz
[Finished in 186ms]
```

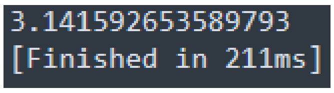
Import libraries:

Using libraries/modules let us build on top of what others have already done. Let's take an example of printing out the value of Pi, we will use a library called "math" to do this. A library is imported in Python by using the keyword `import`.



```python
import math
print(math.pi)
```

The output will be:



```
3.141592653589793
[Finished in 211ms]
```

In the next week, we will be using this feature very frequently since we will work with Tensorflow/Pytorch frameworks to build Machine Learning & Deep Learning algorithms.

## 2. Google Colab introduction and walkthrough

Google Colab is a Python computing environment that enable you to run Python code interactively in the same space as written text. In Google Colab, all necessary libraries for Machine Learning are already downloaded and configured for you. You just need to import them to use. They also provide free GPU/TPU access to train your Deep Learning models.

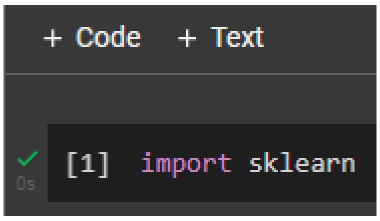
In the file named "Google Colab introduction", we have given a detailed instruction on how to start with Google Colab and perform some basic functions. Please take a look to have a clearer view about Google Colab and how to use it as it is very useful for those of you want to do the Machine Learning/Deep Learning projects.

## 3. Scikit-learn (sklearn)

Sklearn is the oldest machine learning library in the trio, hence it only supports old algorithms and just a few new machine learning models. It provides a selection of efficient tools for machine learning and statistical modelling including classification, regression, clustering and dimensionality reduction via a consistence interface in Python.

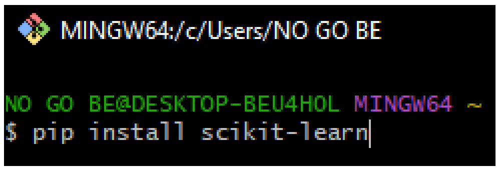
<u>Using sklearn:</u>

If you are using Google Colab, you do not need to install sklearn because it is already installed in Colab. You just need to import to use it:



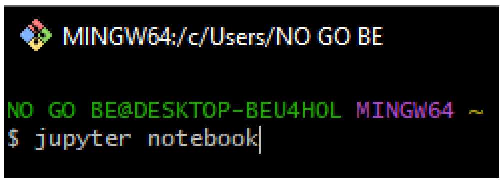
```
[1] import sklearn
```

If you are running Jupyter Notenook on your local machine, you need to install sklearn via pip:



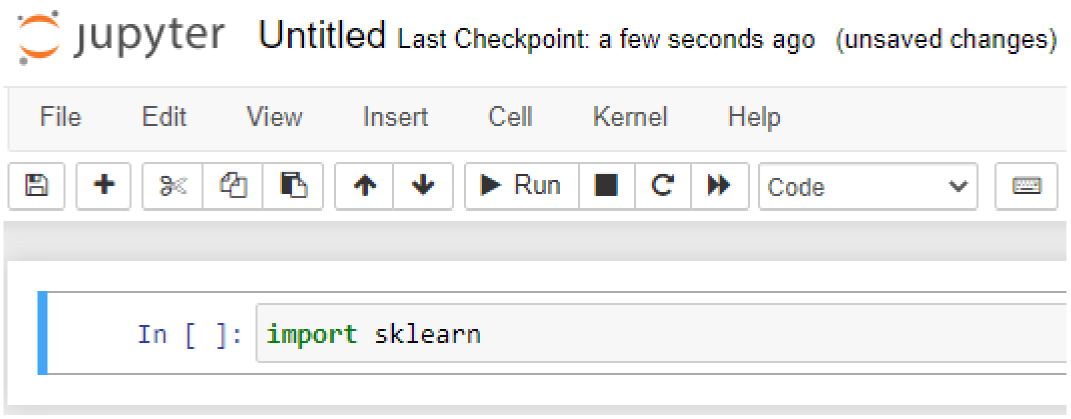
```
MINGW64:/c/Users/NO GO BE

NO GO BE@DESKTOP-BEU4HOL MINGW64 ~
$ pip install scikit-learn
```

Then, you can open your Jupyter Notebook by typing the following in the command line:



```
MINGW64:/c/Users/NO GO BE

NO GO BE@DESKTOP-BEU4HOL MINGW64 ~
$ jupyter notebook
```

Create a new notebook and now you can start using sklearn by importing it like this:


jupyter
Untitled Last Checkpoint: a few seconds ago (unsaved changes)
File
Edit
View
Insert
Cell
Kernel
Help
Run
Code

```
In [ ]: import sklearn
```

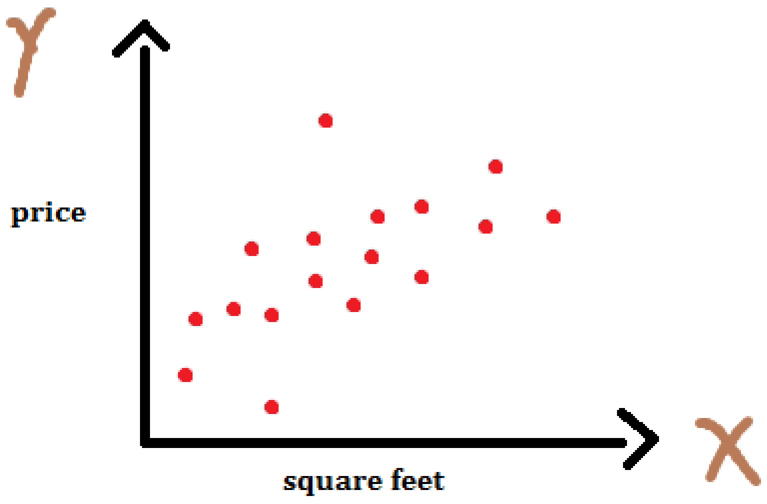
## 4. Linear Regression implementation

We will start our Machine Learning journey with a very simple, yet powerful algorithm called Linear Regression. There are 3 types of Machine Learning algorithms: Supervised Learning, Unsupervised Learning, and Reinforcement Learning. In the Supervised Learning category, problems can be divided into two types of problems- *Regression* and *Classification*. As the name suggests, linear regression is used for regression problems, that is, searching for relationships among variables.

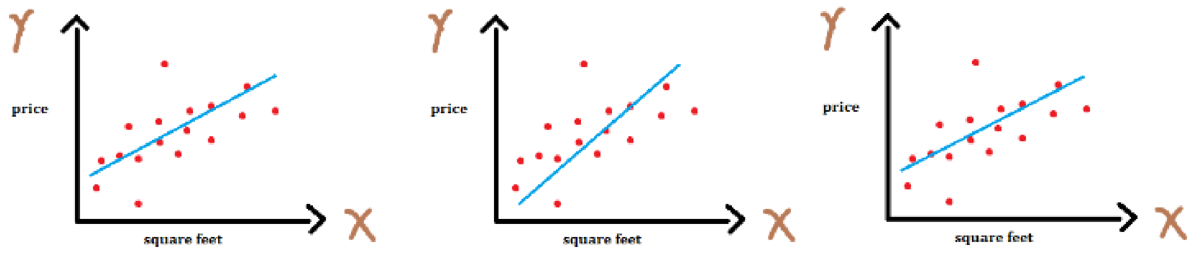
For example, suppose we have a dataset containing house area and the price of the house. Linear Regression can find the relationships between these 2 variables and make price predictions on unseen data. Let's implement a Linear Regression model to do so. Here is how our dataset looks like:

| Area (sq.ft) | Price (1000$s) |
|---|---|
| 3456 | 600 |
| 2089 | 395 |
| 1416 | 232 |

In linear regression, our task is to establish a relationship between target variable and input variables by fitting a line. This line is known as regression line. This line can be represented as a linear equation $\boldsymbol{y} = \boldsymbol{m} \times \boldsymbol{X} + \boldsymbol{b}$. Where, y - target variable, X - input data, m - slope, b - intercept. An intercept represents the intersection of regression line with y axis. Now we can plot our dataset as follows:



Apparently, we can fit n number of lines by tweaking coefficients m and b as shown below.



We can rewrite our equation as $y(x) = w_0 + w_1 * x$ where, $w_0$ is the bias term and $w_1$ is the slope. Here, $w_i$ are known as the weights or parameters of our model. This equation can be used when we have one input variable (feature or independent variable) and this is called as simple linear regression. But this is not the ideal case. We usually deal with the datasets which have many features, the case when we have more than one features it is known as multiple linear regression. We can generalize our previous equation for simple linear regression to multiple linear regression as:

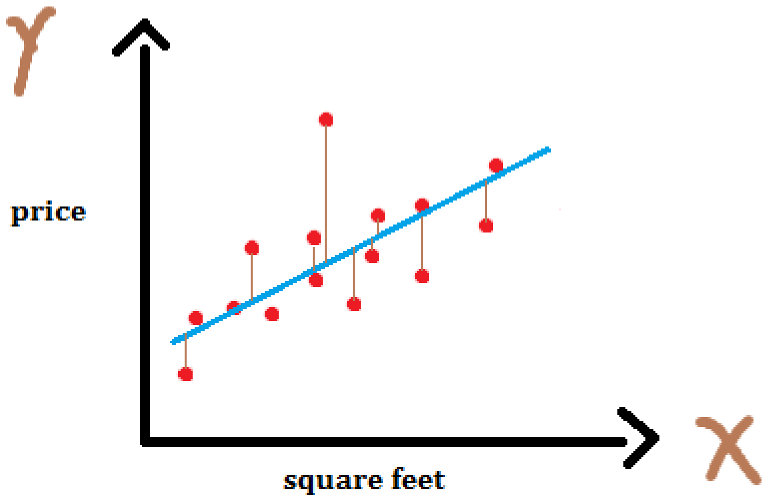
$$y(x) = w_0x_0 + w_1x_1 + \cdots + w_nx_n$$

Where $x_0 = 1$ (intercept term) and $w_0$, as mentioned earlier, is the $y$-axis intercept.

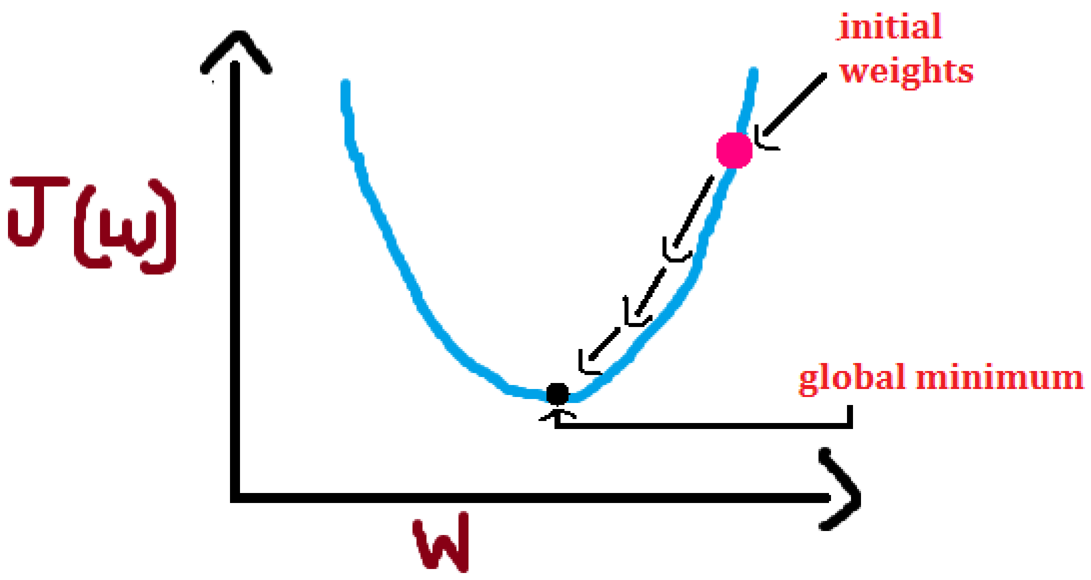
By changing the weights, we can get different lines and now our task is to find weights from which we will get the best fit line. One question you might have is, how can we determine how bad or good a particular line fits our data? For this, we introduce a cost function which measures, for each set of weights $w_i$, how close the $y_{predicted}$'s are to the corresponding $y_{true}$'s. We define our cost function as simple sum of squared error. Here, 1/2 is added to make derivation easy as we will see later.

$$J(w) = \frac{1}{2}\sum_{i=1}^{n}\left(y(x^i) - y_t^i\right)^2$$

Basically, our cost function calculates the distance between true target and predicted target which is shown in the graph as lines between sample points and the regression line below:



For each value of weight, there will be some cost or an error. We want to find the value of weights for which cost is minimum. We can obtain those weights by Gradient Descent:



Gradient descent can be used to minimize the cost function $J(w)$ by updating the parameters/weights in the opposite direction of the gradient of the cost function $\nabla_w J(w)$ with respect to the parameters. Mathematically we can write as:

$$w = w - \alpha \cdot \nabla_w J(w)$$

Where $\alpha$ is the learning rate which determines the size of the steps we take to reach a minimum. We need to be very careful about this parameter since high value of $\alpha$ may overshoot the minimum and very low value will reach minimum very slowly.

Let’s see how Linear Regression can be implemented in Python with “sklearn” library in the notebook “Linear_Regression.ipynb”.