



# Python

- Python is a widely used high-level programming language for general-purpose programming, created by **Guido van Rossum** and first released in 1991.
- Guido van Rossum was big fan of **Monty Python's Flying Circus** and hence had given the name of this programming language as Python.
- Python uses a syntax that allows programmers to express concepts in fewer lines of code than might be used in languages such as C++ or Java.



# Features of Python

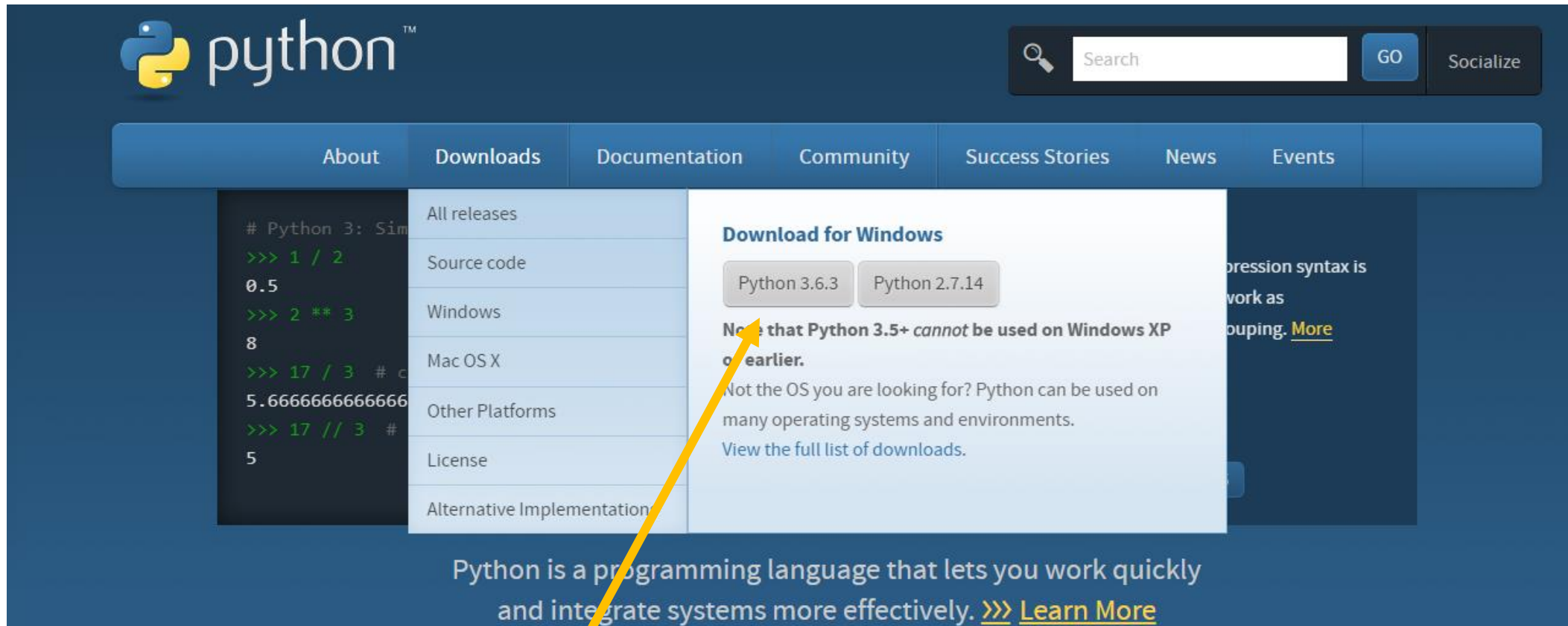
- Easy-to-learn** – Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- Easy-to-read** – Python code is more clearly defined and visible to the eyes.
- Easy-to-maintain** – Python's source code is fairly easy-to-maintain.
- Interactive Mode** – Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- Extendable** – You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- Databases** – Python provides interfaces to all major commercial databases.
- Scalable** – Python provides a better structure and support for large programs than shell scripting.



# Installation of Python 3.6.3

## Steps for Installing Python

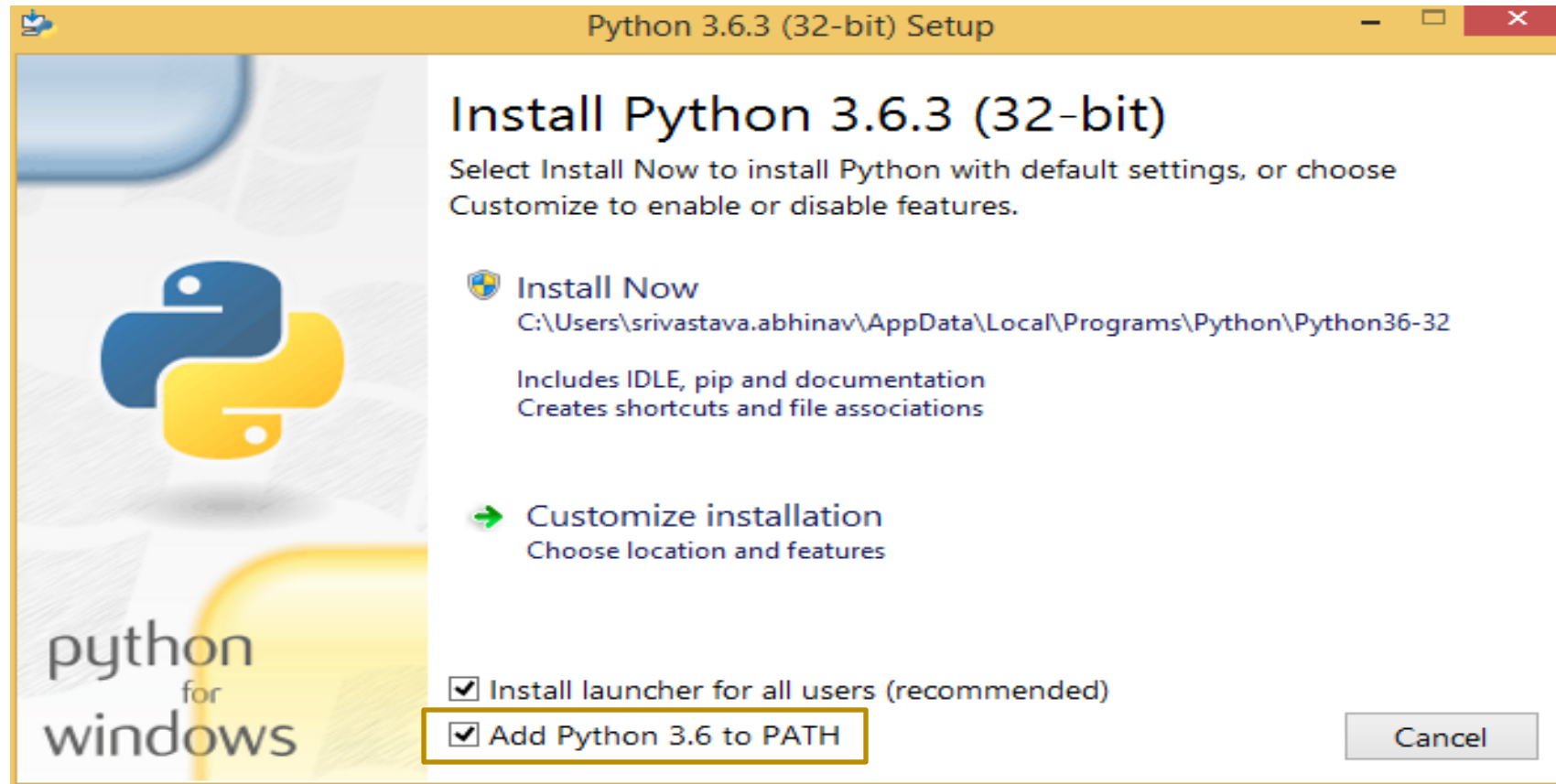
1. Download the latest Python build from [www.python.org](http://www.python.org) as shown in Figure 1 below.



When you double click on Python 3.6.3, Python installer file will be downloaded in “Download” folder.

# Installation of Python 3.6.3

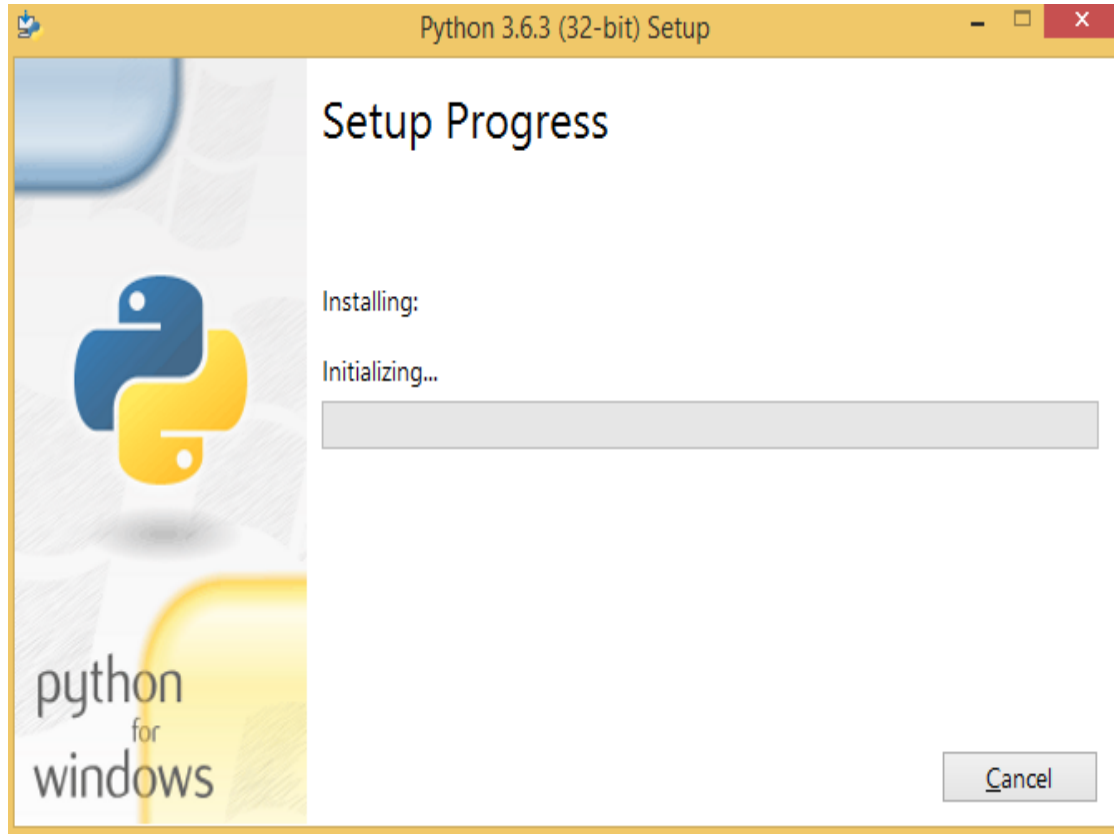
2. Double click the downloaded Python installer file to start the installation. You will be presented with the options as shown in Figure 3 below. Make sure that "Add Python 3.6 to PATH" checkbox is ticked.



Click the Install Now and Python 3.6.3 install will proceed.

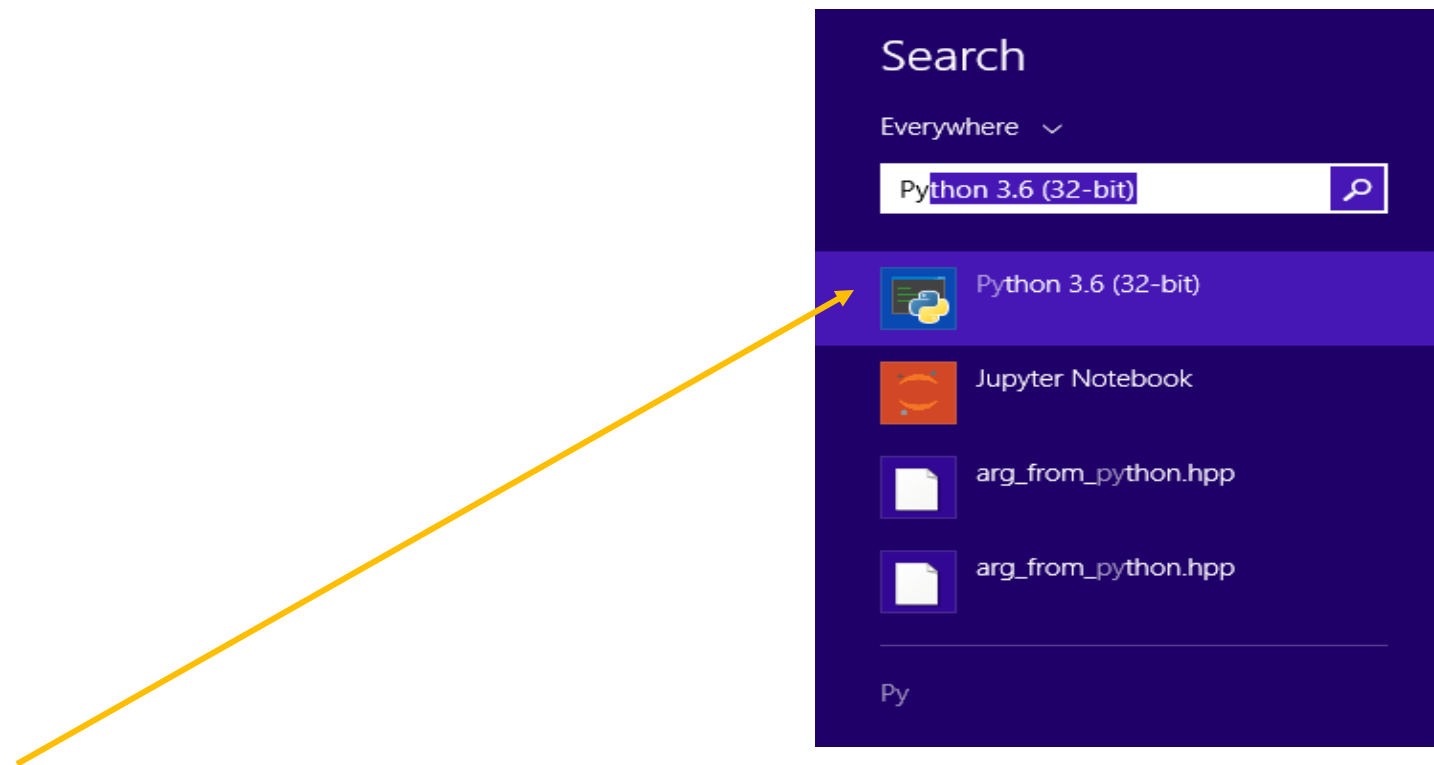
# Installation of Python 3.6.3

This shows the successful installation of Python 3.6.3



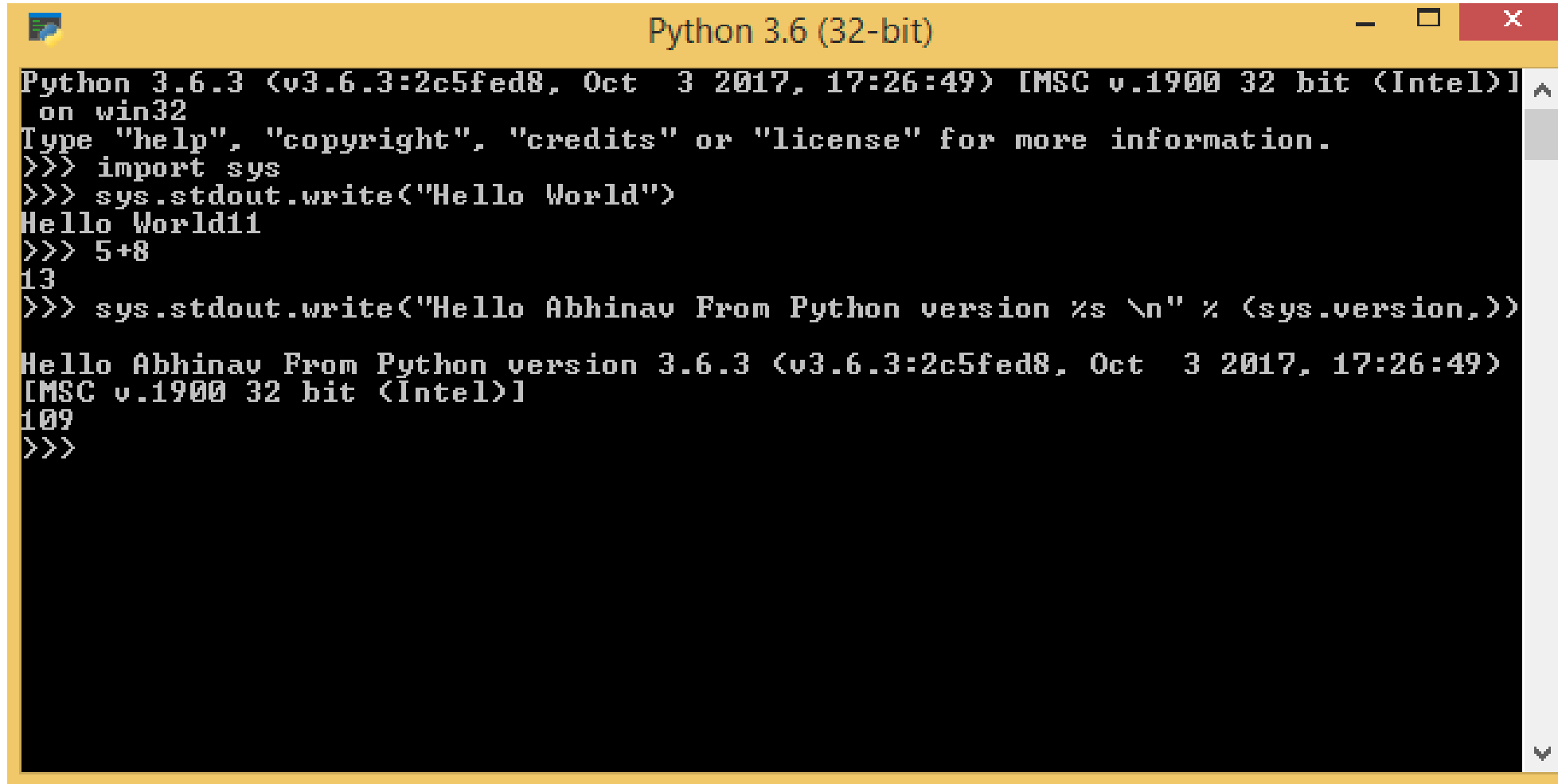
# Installation of Python 3.6.3

3. Once the installation has successfully completed, Python 3.6.3 will be installed and available from the Windows Search Menu. Run Python 3.6 (32-bit) from the start menu. It can be found in the Python folder as shown in Figure below.



Click on Python 3.6 and it provides an interactive console that gives a way to immediately write Python code and see the results in real time – Shown in next slide

# Installation of Python 3.6.3

A screenshot of a Windows command prompt window titled "Python 3.6 (32-bit)". The window has a yellow title bar and standard Windows window controls (minimize, maximize, close). The command prompt shows the following text:

```
Python 3.6.3 (v3.6.3:2c5fed8, Oct 3 2017, 17:26:49) [MSC v.1900 32 bit (Intel)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.stdout.write("Hello World")
Hello World!
>>> 5+8
13
>>> sys.stdout.write("Hello Abhinav From Python version %s \n" % (sys.version,))

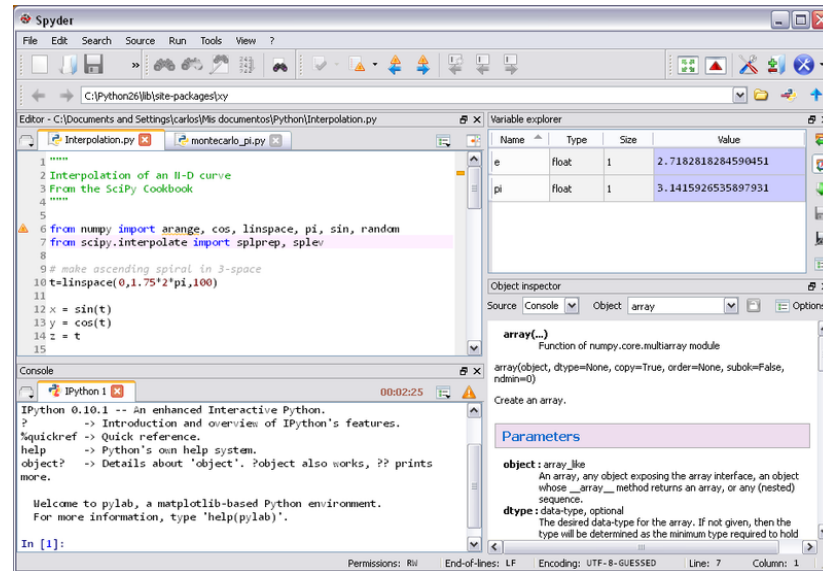
Hello Abhinav From Python version 3.6.3 (v3.6.3:2c5fed8, Oct 3 2017, 17:26:49)
[MSC v.1900 32 bit (Intel)]
109
>>>
```



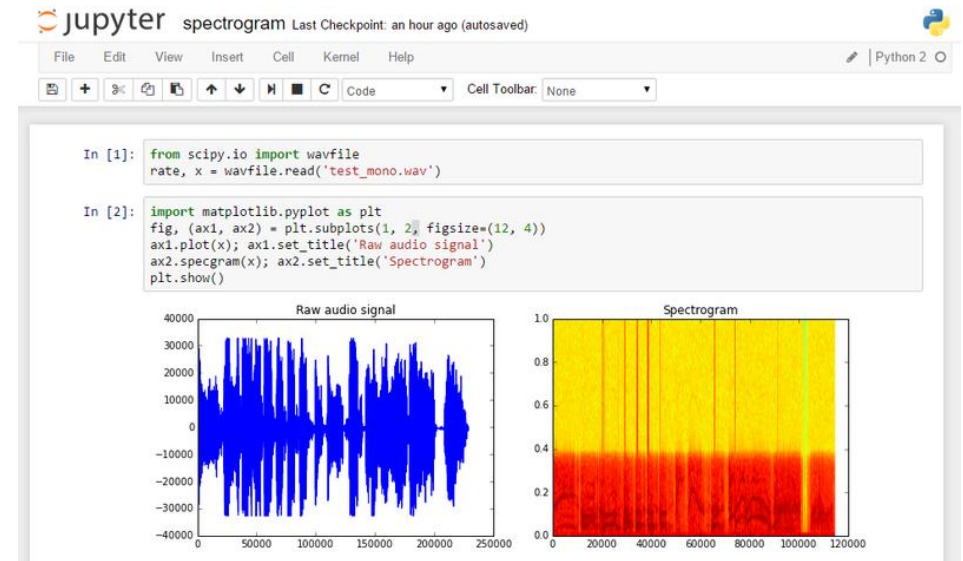
# IDE for Python -> Integrated Development Environment

It's a coding tool which allows you to write, test and debug your code in an easier way, as they typically offer code completion or code insight by highlighting, resource management, and debugging tools.

- Spyder
- Jupyter
- Atom
- Pycharm
- Rodeo



Spyder

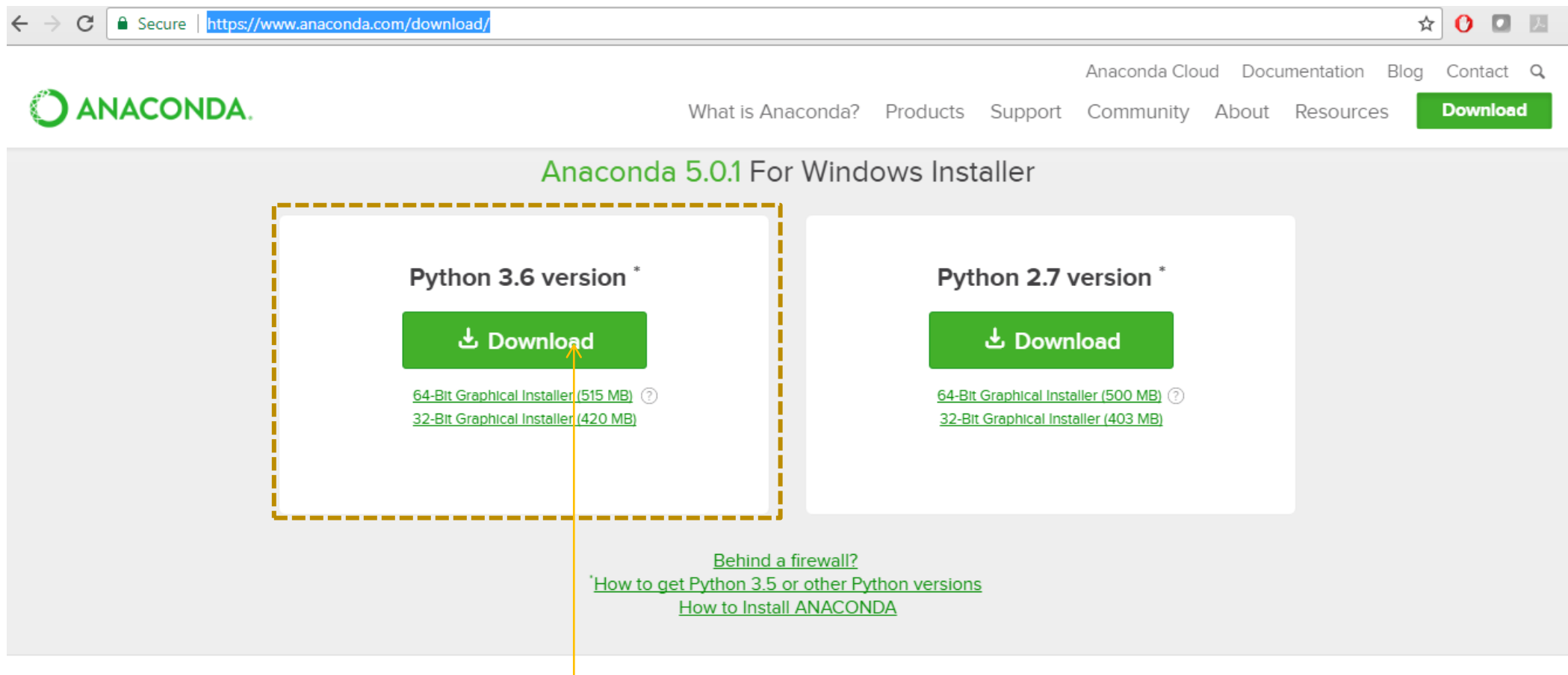


Jupyter

# Anaconda Installation – Many problems one solution

1. Conda is an open source package management system and environment management system that runs on Windows, macOS and Linux.
2. Conda quickly installs, runs and updates packages and their dependencies.
3. Conda easily creates, saves, loads and switches between environments on your local computer.
4. It was created for Python programs, but it can package and distribute software for any language.
5. Conda as a package manager helps you find and install packages.
6. The conda package and environment manager is included in all versions of Anaconda.
7. If we install Anaconda – **Spyder**, **Jupyter**, and even **Python** latest version would also be installed simultaneously.
8. To install Anaconda -> Click on to the following links  
<https://www.anaconda.com/download/>
9. Anaconda package which provides code completion and linking for Python and takes care of the basics, there are tons of themes, lightning fast user interface, easy configuration, tons of packages for more power features

# Anaconda installation



← → ↻ Secure <https://www.anaconda.com/download/> ☆ ⓘ 📄 🗑️

ANAACONDA.

Anaconda Cloud Documentation Blog Contact 🔍

What is Anaconda? Products Support Community About Resources **Download**

## Anaconda 5.0.1 For Windows Installer

### Python 3.6 version \*

**Download**

[64-Bit Graphical Installer \(515 MB\) ?](#)  
[32-Bit Graphical Installer \(420 MB\)](#)

### Python 2.7 version \*

**Download**

[64-Bit Graphical Installer \(500 MB\) ?](#)  
[32-Bit Graphical Installer \(403 MB\)](#)

[Behind a firewall?](#)  
[How to get Python 3.5 or other Python versions](#)  
[How to Install ANACONDA](#)

To download Anaconda click here and install it. Jupyter, Spyder and python will be installed simultaneously.

## 3.1 Python Identifier

- Python identifier :-
  - Identifier is the name given to entities like class, functions, variables etc. in Python. It helps differentiating one entity from another. It can be of any length.
  - Starts with a letter A to Z or a to z or an underscore (\_) which is used to identify a variable, function, class, module or other object .
  - An identifier cannot start with a digit. 1variable is invalid, but variable1 is perfectly fine.
  - Keywords cannot be used as identifiers.
  - We cannot use special symbols like !, @, #, \$, % etc. in our identifier.
  - Every class name start with capital letter.
  - Strange and strange are two different identifiers in python.

Strange

starts with uppercase.

strange

starts with lowercase.

## 3.2 Python Keywords

- Python keywords are mentioned below. The following are keywords of python.

<b>False</b>	<b>Class</b>	<b>finally</b>	<b>is</b>	<b>return</b>
None	continue	from	lambda	while
True	def	for	nonlocal	try
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

## 3.3 Lines And Indentation

- Blocks of code are denoted by line indentation .
- The two blocks of code in our example if-statement are both indented four spaces, which is a typical amount of indentation for Python

```
In [1]: x=10  
        if(x>1):  
            print("Big")  
        else:  
            print("Small")
```

Big

Here , there is four space indentation. It is required to execute else there would be error.

All the continuous lines indented with same number of spaces would form a block .

# Generates Error

```
In [2]: x=10  
if(x>1):  
print("Big")  
else:  
print("Small")
```

```
File "<ipython-input-2-0f22b094bf2f>", line 3  
    print("Big")  
    ^
```

**IndentationError:** expected an indented block

Here , we have not indented four space, and written same code. So, "Indentation Error" takes place.

## 3.4 Multi Line Statements

- By using multi-line statements we can continue a line.

Example :

```
In [6]: x=10  
if x == 10 or x > 0 or \  
    x < 100:  
    print('True')
```

True

```
In [8]: def print_something():  
        print ('Wow, this also works?',  
              'I never knew!')  
print_something()
```

Wow, this also works? I never knew!

“\” is used as continuation of line. It is used universally (at any place)

“,” I used here as continuation of line, but it is applicable only with print statement.

Statements which are in [], {}, or () brackets do not use the line continuation character.

For example:

```
In [17]: week = ['Monday', 'Tuesday', 'Wednesday',  
                'Thursday', 'Friday']  
print(week)
```

['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']



## 3.5 Quotation in Python

- Python accepts single ('), double (") and triple (''' or ''')quotes to denote string literals.
- Examples :
  - 'apple' ,
  - "welcome to python",
  - ''' This is another form to accept as string literals'''

## 3.6 Comments in Python

- There are two types of comments in python.

### 1. single line comment

- It begins with the hash character ("#") This is the only way to comment on Python

### 2. Multiple line comment

- It doesn't exist in python.

### 3.Waiting for the User

- The statement saying “Press the enter key to exit”, this means it waits for the user’s action.

### 4.Multiple Statements on a Single Line

- The semicolon ( ; ) allows multiple statements on the single line .

### 3.7. Multiple statement groups as suites

A compound statement consists of one or more 'clauses.' A clause consists of a header and a 'suite.' The clause headers of a particular compound statement are all at the same indentation level. Each clause header begins with a uniquely identifying keyword and ends with a colon. **A suite is a group of statements controlled by a clause.** A suite can be one or more semicolon-separated simple statements on the same line as the header, following the header's colon, or it can be one or more indented statements on subsequent lines.

```
if_stmt ::= "if" expression ":" suite ←  
          ( "elif" expression ":" suite ) *  
          ["else" ":" suite]
```

If condition is met, then clause is executed  
Each such clause is suite.

### 3.8. Command Line Arguments

An argument sent to a program being called. More than one command lines are accepted by the program.

```
In [20]: import sys
print ("This is the name of the script: ", sys.argv[0])
print ("Number of arguments: ", len(sys.argv))
print ("The arguments are: " , str(sys.argv))
```

This is the name of the script: C:\Users\hi\Anaconda3\lib\site-packages\ipykernel\_launcher.py  
 Number of arguments: 3  
 The arguments are: ['C:\\Users\\hi\\Anaconda3\\lib\\site-packages\\ipykernel\_launcher.py', '-f', 'C:\\Users\\hi\\AppData\\Roaming\\jupyter\\runtime\\kernel-3d70376c-310f-48a7-93b4-bc20b7a6ddbc.json']

sys.argv is a list in Python, which contains the command-line arguments passed to the script.

With the len(sys.argv) function you can count the number of arguments.

If you are going to work with command line arguments, you probably want to  
use sys.argv.

To use sys.argv, you will first have to import the **sys module**.

# Assigning Values to Variables

- The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

- $H = 5 + 5$

H

10

- **Multiple Assignment**

- Here one is allotted to multiple variables. We can assign objects to multiple variables.
- $x = y = z = 1$

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

# Python Arithmetic Operators

- We use these operators for mathematical purpose.

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	$x + y = 40$
- Subtraction	Subtracts right hand operand from left hand operator	$x - y = -20$
* Multiplication	Multiplies values on either side of the operator	$x * y = 300$
/ Division	Divides left hand operand by right hand operand	$y/x = 3$
% Modulus	Divides left hand operand by right hand operand and returns remainder	$x \% y = 0$



# Python Operator Precedence

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division + - Addition and subtraction

- We use these operators for conditional purpose.

Operator	Description ( usage)
>	if the left operator has greater value than the right one. Ex= 2>1
<	If the right operator has greater value than the left one. Ex= 6<9
==	If both sides of the operators are same to same . Ex= 4==4
!=	if both sides of the operators are not same.
>=	If the left operator is greater than or equal to right one. Ex = 6>=6,7>=4
<=	If the right operator is greater than or equal to left one. Ex= 4<=4,6<=8

- These are used to assign values to variables. Few one them are as shown below.

Operator	Description and Example
=	S =2+3 means the value 2+3=6 is assigned into S
+=	H+=4 means 4 is added to H and final result is saved into H. So H+4=H. i.e if H value is 2 then 6+4=10 is stored in H.
-=	H-= 4 means 4 is subtracted to H and final result is saved into H. So H-4=H i.e 6-4=2 is stored in H.
*=	H*= 4 means 4 is multiply with H and final result is saved into H. So H*4=H i.e 6*4=24 is stored in H.

- Similarly few more operators like %=, //=, \*\*=, &=, |= .....

- Logical operators are also used to check conditional statements.

Operator	Description
AND	used to check both the operands or true  Ex a=2,b=2 if a AND b=2 then print(' a AND b is', True) i.e output is TRUE
OR	Used to check either one of the operator is true Ex a=2,b=3 if a OR b =3 then print(' a OR b is ', 3) i.e output is b=3.
NOT	Used to complement the operator If given X is true then it shows it is false .

- These operators are used to check whether the given values are in the sequence or not.

Operator	Description
in	If the given value is within the sequence it will displays . Example as shown below X = { 2,4,6,8} Print(4 in X ) then it will display 'True' because 4 is within the x
Not in	Opposite to IN operation. Example : 9 Not in then it will check whether 9 is in sequence or not. Regarding to the above sequence we do not have 9 so it will display 9.

- These operators are used to check whether the values are in the memory or not.

Operator	Description
is	If the operand are same then it is true. EX: x=2,y=2 then print X is Y.
Is not	If the Operand are not same then it shows true. Ex: x=2 ,y=2 print x is not Y then it shows false. Ex: x=2y=3 print x is not y then it shows True.

- Variable is used to store data and to retrieve the data. That means, we can change its value .

It is not fixed. Example

```
In [22]: x = 12; y = 10;  
         c = x + y  
         print("The value of c when x is 12 : ",c)  
         x = 10  
         c = x + y  
         print("The value of c when x is 10 : ",c)  
  
The value of c when x is 12 : 22  
The value of c when x is 10 : 20
```

Initially the value of x is 12, so it gives 22  
After we changed it to 10, so it gives 20 .Hence we can change the values.

In python variables are case sensitive and consists of numbers, letters and underscore.

- Reserved words should not be used as a variables names.

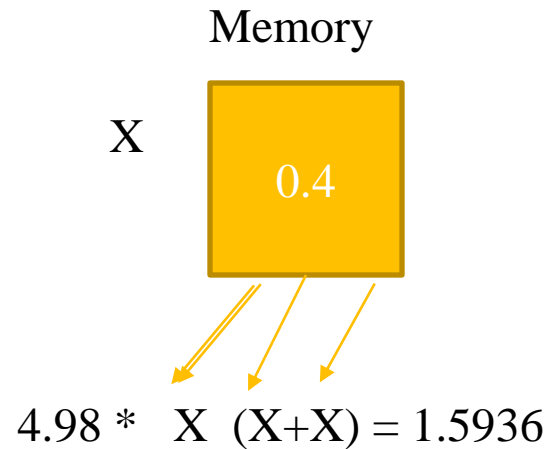
Example : else , return ,and print etc.

- As we know that variable is used to store a value lets take one example

$$X = 0.4$$

That means X value is stored as 0.4

solve  $X = 4.98 * X * (X + X)$  then the operation done in memory is as follows.1





- Now  $X = X - 0.2$   
Ans is 1.3936



Now the x value is changed to 1.5936 why because we just find it by using the previous expression so like that it is updated

# Operator Precedence



- During operation where one operand is an integer and the other operand is a floating point the result is a floating point
- The integer is converted to a floating point before the operation.

- We use conditional statements for checking the condition of the statement and to change its behavior.
- Some of the conditional statements are as follows...
  1. if
  - 2.if else
  - 3.Nested if else

Now lets discuss about If statement.

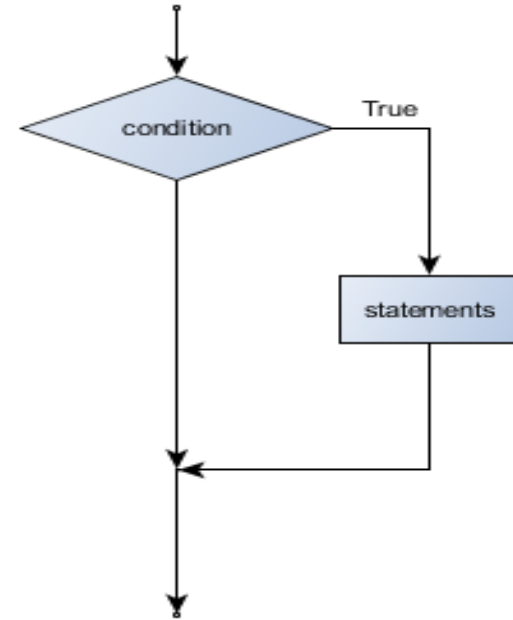
1.If

Syntax for if is as follows

if expression:

statement(s)

- A flowchart is given for better understanding ..
- The boolean expression after the if statement is called the **condition**. If it is true, then all the given statements are executed . Otherwise no
- Example as shown below.



```
In [23]: item = "jalebi"
         if item == "jalebi":
             print("I am loving it")
```

I am loving it

Here there is an expression "=" which tells that variable is assigned some value.

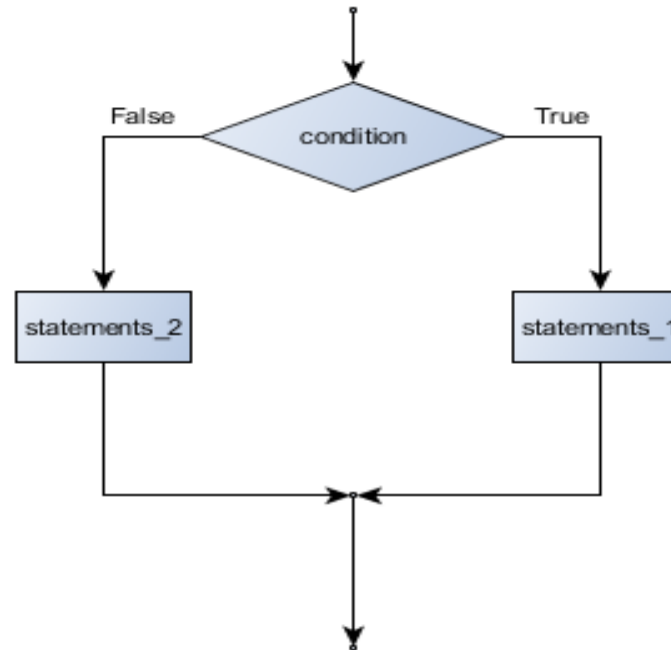
Here there is an expression "==" which tells that there is a condition

The statement is executed as the condition was true.

## 2.If Else

- One thing to happen when a condition is true, otherwise **else** to happen when it is false.
- Syntax:
  - if expression:
    - statement(s)
  - else:
    - statement(s)
- The below flowchart describes how if else condition works

- If the given condition is true then statement one will display other wise second statement.



## Example

```

In [25]: item = "dal"
         if item == "jalebi":
             print("I am loving it")
         else:
             print("I just want jalebi")
  
```

I just want jalebi

Here the "if" condition hasn't met, so "else" statement has been executed

# Nested If Else

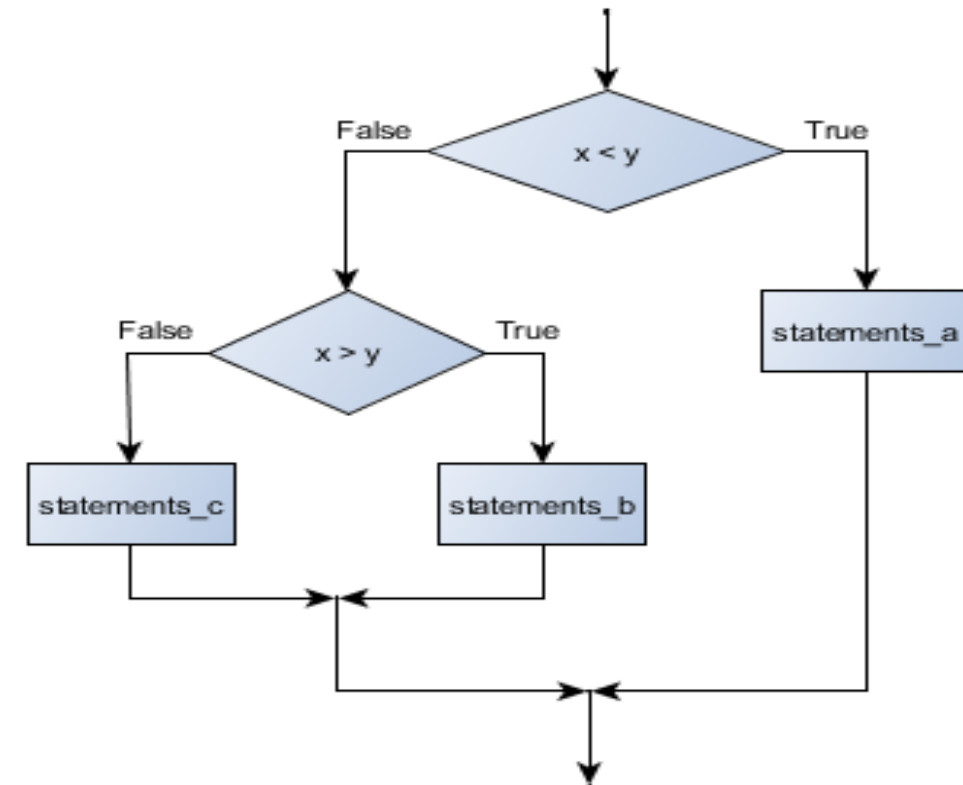
- If the first condition is true(  $x < y$ ), it displays the first one. If it is false, then checks second condition ( $x > y$ ) if it is true then displays second statement . If the second condition is also not satisfied, then it displays the third statement.

- Example :

```
In [28]: item = "dal"
if item == "jalebi":
    print("I am loving it")
elif item == "dal" :
    print("I need rice also")
else:
    print("I will not eat")
```

I need rice also

To use nested else if in python, we have a key word "elif". It will be executed after "if", and before "else"

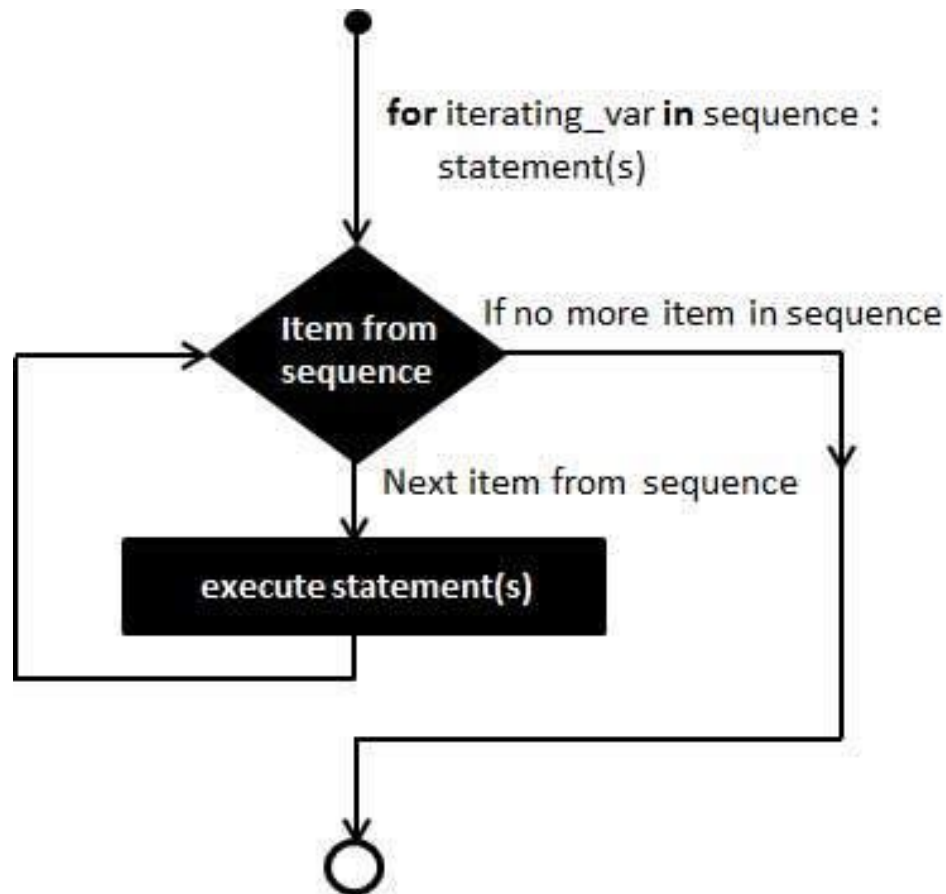


- Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. It is a continuous process.
- Syntax:
  - for iterating\_var in sequence:  
    statements(s)
- The below flow chart gives how for loop works



# For Loop

- Each iteration assigns the loop variable to the next element in the sequence, and then executes the statements in the body. Statement will be completed if it reaches the last element in the given sequence.



- Example :

```
In [29]: for letter in "DARLING":  
         print("current letter : ",letter)
```

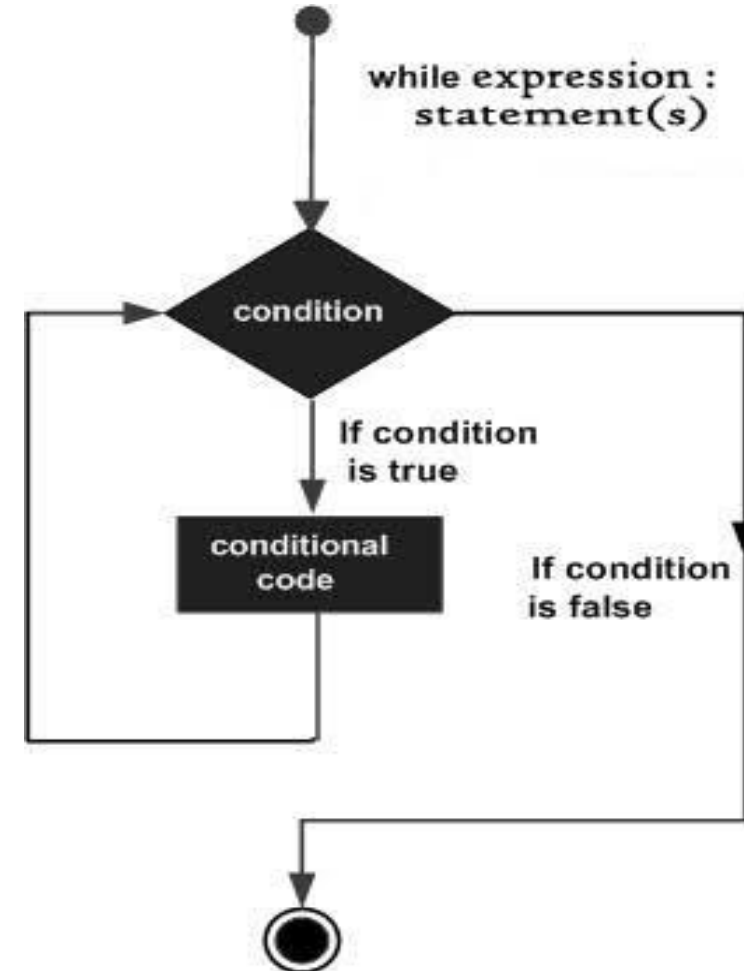
```
current letter : D  
current letter : A  
current letter : R  
current letter : L  
current letter : I  
current letter : N  
current letter : G
```

To use "for " loop, we have to use keyword "in"

Under variable "letter", every letter will pick one by one in each loop.

# While Loop

- while loop executes repeatedly the target statement as long as a the given condition becomes true.
- Syntax
  - while expression:  
statement(s)
- Flow chart gives how does the while loop works.



- Example

```
In [36]: count = 4
while (count > 0):
    print('The count is:', count)
    count = count - 1
print("Good bye!")
```

The count is: 4  
The count is: 3  
The count is: 2  
The count is: 1  
Good bye!

“while” loop continues or repeats its conditions until its last statement become true. Here loop will run, until “count” will become equal to or less than zero.

Last “print” statement will be executed out of “while” loop because it is not have four space indentation

# break Statement

- It terminates the current **loop** and resumes execution at the next **statement**.

## •Example:

```
In [39]: for letter in "Welcome":  
        if letter == "l":  
            break  
        print ('Current Letter :', letter)
```

Current Letter : W  
Current Letter : e

“break” statement breaks the loop if “if” condition comes true.

# continue Statement

- It returns the control to the beginning of the while loop.. The **continue statement** rejects all the remaining **statements** in the current iteration of the loop and moves the control back to the first of the loop.

```
In [40]: for letter in "Welcome":  
         if letter == "l":  
             continue  
         print ('Current Letter :', letter)
```

```
Current Letter : W  
Current Letter : e  
Current Letter : c  
Current Letter : o  
Current Letter : m  
Current Letter : e
```

“continue” statement allow the loop to skip the step, if “if” condition comes true.

- Function is a collection of statements which is used to perform a specific task.
- A method is a function that takes a class instance as its first parameter
- Functions in Python are similar to functions in “C” and functions/procedures in “Pascal”.

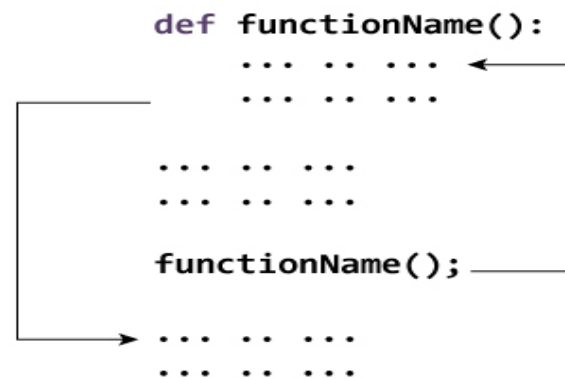
- *Syntax*

```
def function_name(parameters):  
    """doc string"""  
    statement(s)
```

- Function begins with *def* Keyword followed by function name and parentheses
- Function naming follows the same rules of writing Python identifiers.

- We pass values through parameters or arguments and these are placed within the parenthesis.
- Colon is used to end the function header.
- Documentation string( *doc string* ) is used to explain what the function does.
- And optional return statement to return value from function.

*How Function works in Python.*





- Example 1:

```
In [43]: def add( p,q):
         return p+q

         print("first return : ",add(4,6))
         r=add(3,2)
         print("second return : ",r)

first return :  10
second return :  5
```

“add” function is created, sum of two parameters is returned in this function

- Example 2:

```
In [46]: def greet(name):
         print("Welcome "+name+" to the Python world")
         greet("Abhinav")

Welcome Abhinav to the Python world
```

To call a function, simply use the function name with apt parameters or arguments. Here the apt parameter is name(string). If string variable would not be provided, error will come

# Function With parameters

- Parameters or arguments are used for passing values. Immutable objects cannot be changed inside the functions or outside. But mutable objects or opposite to Immutable objects, which can be changed outside the function.
- Example :

```
In [50]: def swap(a,b):  
         return(b,a)  
a=10  
b=20  
print("original value : ",a,b)  
a,b=swap(a,b)  
print("swapped value : ",a,b)  
  
original value : 10 20  
swapped value : 20 10
```

“swap” function is created, to swap the values of parameter.

Value is assigned to variables from “swap” function, which is returned by it

First print statement returned “original value” i.e. the value which is assigned initially.

After that swap statement swaps the value, so then b value is changed to a and vice versa.

# Function without parameter

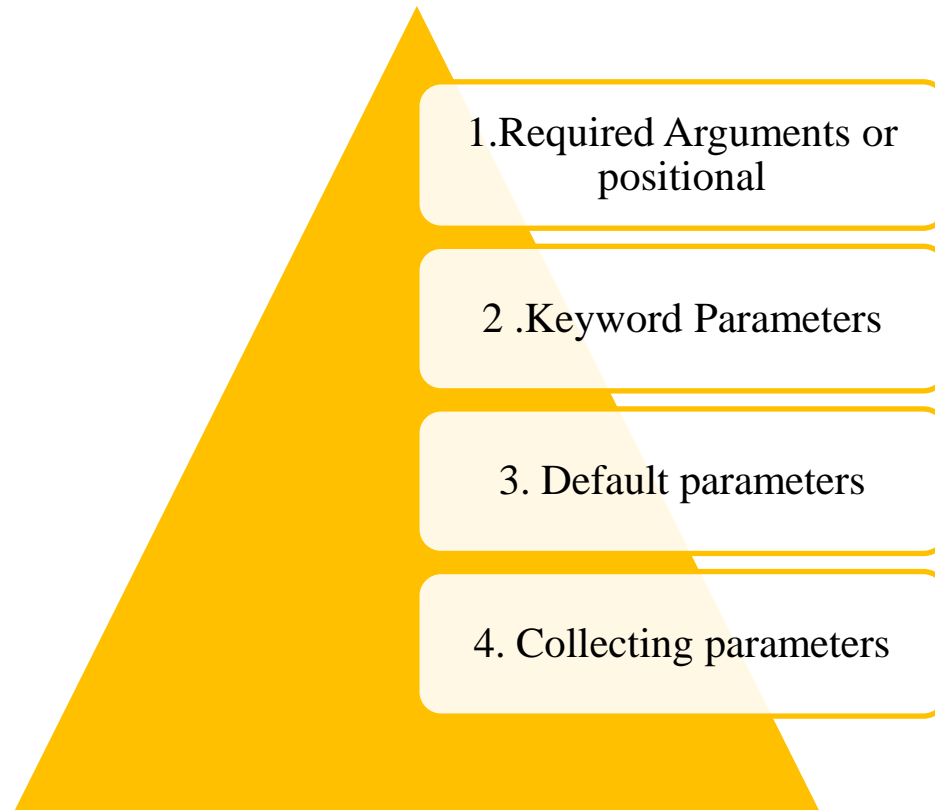
There is no compulsion for a function to be associated with parameters or arguments.

```
In [45]: def miss():  
         print("food is missed")  
         food="rice"  
         if food == "curd":  
             print("here is your food")  
         else:  
             miss()  
  
food is missed
```

"miss" function is without parameter.  
It will return a constant value which is kept under the function.

"miss" function is called here. It would be more practical to use, when we have to print similar statements multiple times. In such case, each time, we have to just call the function, in place of writing complete print statement.

- In functions there are four parameters they are as follows



# 1.REQUIRED ARGUMENTS OR POSITIONAL PARAMETERS

- The arguments or parameters which are in a positional format in a( parameters must be in a order) function is nothing but Positional Arguments or parameters. Parameters in function call will perfectly match with the function definition.
- Example

```
In [17]: def num(val1,val2,val3, val4):  
         return val1 * val2 *val3*val4  
  
         print(num(4,2,6,1))
```

48

Here we had given 4 positions so when we give values like 4,2,6,1 then they will take those values in a order.

- Given values are 4,2,6,1 those can be placed in order by these positional argument.

Val 1	val2	val3	val4
4			
4	2		
4	2	6	
4	2	6	1

- First value 4 will enter into val1 and then second value will be to val2 so on ...

## 2 .KEYWORD PARAMETERS

- We can identify the function call as Keyword argument by its argument name.
- In this argument no need of maintaining the parameters in an order.
- Example :

```
In [9]: def printdtl( product, cost):  
        # "This prints a passed info into this function"  
        print("Product :", product)  
        print("Cost:", cost)  
        return;
```

The "product" and "cost" are keyword parameters.

```
In [10]: # Now you can call printdtl function  
         printdtl (cost=2500, product="keyboard")
```

The value of the parameter is taken from function in different order as declared in the function.

```
Product : keyboard  
Cost: 2500
```

### 3. DEFAULT PARAMETERS

- Default parameter is nothing but parameter which is assigned default when value is not assigned in a argument when function is called.
- These values will evaluate once the function is called or defined.
- Example :

```
In [8]: def sal_HRA (x, HRA = 4):  
        return x*(HRA/100)
```

```
In [14]: print (sal_HRA (10000))  
400.0
```

```
In [15]: print(sal_HRA(20000))  
800.0
```

Here HRA is default parameter.  
HRA(given as percent) to value x is  
returned



## 4.COLLECTING PARAMETERS

- Sometimes it can be useful to allow the user to supply any number of parameters.
- Actually that's quite possible with the help of collecting parameters

```
In [4]: def print_params(*params):  
        print(params)
```

```
In [5]: print_params(1,2,3)  
  
(1, 2, 3)
```

```
In [6]: print_params('key')  
  
('key',)
```

Here parameter is passed through (\*) which allows user to input any number of parameters

Here 3 values to the parameter is passed which is accepted by function

- All variables cannot be accessed at a same place or at all locations. They are accessed at different places .
- Usually they are some Scopes for allocating variables at a particular location.
- They are :

## LOCAL VARIABLES

- Variables that are accessed inside the function is called Local Variables.

## GLOBAL VARIABLES

- Variables that are accessed throughout the function or outside the program is called Global Variables.

- If we want to search where the variable location is allocated, first search will be in Local variable if it is not found in local variable then the search will be in Global variable . If the variable still cannot be found in Global variable then search will be *in Built in predefined Name space* .
- If still variables are not found there then an exception will be raised i.e. *name is not defined*

```
In [30]: total = 0;      #global variable.  
def mul(arg1,arg2):  
    # multiply both the parameters and return them."  
    total = arg1 *arg2  
    print("Inside the function local total : ", total)  
    # Here total is Local variable.
```

To check the output of local variable :  
its value is printed under function

```
In [31]: mul(6,4)  
  
Inside the function local total : 24
```

```
In [32]: # Now you can call mul function  
print ("Outside the function global total : ", total )  
  
Outside the function global total : 0
```

To check the output of global variable  
: its value is printed outside the  
function

- Return statement can return any type of value which python identifies.
- In Python all functions return at least one value. If they are not defined then there is no return value.

- We can call a function in recursive, i.e. the function will call itself.
- Example :

```
In [33]: def factorial(n):  
         if n == 0:  
             return 1  
         else:  
             return n * factorial(n - 1)
```

```
In [34]: factorial(3)
```

```
Out[34]: 6
```

The function which is created, itself is returned -> Recursive function

- Function defining by using recursion is too costly for executing the program or due its memory location.

- In python there is a possibility to manipulate files .Files are used to store objects permanently. The following are the steps to use Files

1.Printing on screen

2.Reading data from keyboard

3.Opening and closing file

4.Reading and writing files

5.Reading CSV, txt and excel files

# 1.PRINTING ON THE SCREEN

- We can see the statements what we have written in the output by using *PRINT* statement.
- This statement (print) can converts expressions what we are passing into a string and writes
- the result to the standard output.
- *Example* : `print (" Welcome to python")`
- *Output* Welcome to python.

## 2.READING DATA FROM KEYBOARD

- For reading data from keyboard they are 2 built in functions in python.
- They are:
  - Raw input Function.
  - Input function.
- #For version greater than python-3.0

```
In [55]: Num = input("Select a number between 1 to 9 : ")  
         print("The number selected is:" , Num)
```

```
Select a number between 1 to 9 : 7  
The number selected is: 7
```

The “input” function is used to get input from the keyboard.

- #For version less than python-3.0
  - Num = raw\_input(“Select a number between 1 to 9”)
  - print(“The number selected is: “, Num)



# READING AND WRITING FILES

Name	<code>open("filename")</code> opens the given file for reading, and returns a file object
<code>name.read()</code>	file's entire contents as a string

```
In [56]: import os  
         print(os.getcwd() + "\n")
```

```
C:\Users\hi
```

```
In [57]: fobj = open("sample.txt")  
         fobj.read()
```

```
Out[57]: 'it is python '
```

To know where is working directory

Open and read function is used here to read the file

```
In [9]: with open('test.txt') as file:  
        for line in file:  
            print(line)
```

```
it
```

```
is
```

```
python
```

```
multi line
```

```
text
```

To read multiple lines, we have to open it with "with open" keyword and print it in "for" loop

- The `write()` method writes any string to an open file.
- Syntax  
    `fileObject.write(string);`

```
In [65]: fobj = open("sample.txt", 'w')
         fobj.write('to test\n')
         fobj.write('write\n')
         fobj.write('function\n')
         fobj.close()
```

```
In [66]: fobj = open("sample.txt")
         s=fobj.read()
         print(s)

to test
write
function
```

Used the “write” function to write multiple strings in the file. “\n” changes line

- **The open Function:**

- By using Python's built-in open() function we can open a file.

- **Syntax**

- file object = open (file\_name): ([, access\_mode][, buffering])

1. **file\_name:** Name of file.

2. **access\_mode:** The access\_mode determines the mode in which the file has to be opened i.e. read, write

```
In [65]: fobj = open("sample.txt", 'w')
fobj.write('to test\n')
fobj.write('write\n')
fobj.write('function\n')
fobj.close()
```

```
In [66]: fobj = open("sample.txt")
s=fobj.read()
print(s)

to test
write
function
```

Used the access mode 'w' in "open" function in the file

- **The close Function:**
- when the reference object of a file is reassigned to another file. Python automatically closes that file.
- **SYNTAX:**
  - `fileObject.close();`

```
In [65]: fobj = open("sample.txt", 'w')
fobj.write('to test\n')
fobj.write('write\n')
fobj.write('function\n')
fobj.close()
```

```
In [66]: fobj = open("sample.txt")
s=fobj.read()
print(s)

to test
write
function
```

Used the function “close”  
to close the file

# READING CSV, TXT AND EXCEL FILES

- A CSV file contains a number of rows and columns separated by commas.
- **Reading CSV Files**
- To read data from a CSV file, use the reader function to create a reader Object. Use writer() to create an object for writing, then iterate over the rows, using writerow() to print them.

```
In [3]: import os  
        print(os.getcwd())  
C:\Users\hi
```

os.getcwd() helps in getting working directory, while os.chdir is used to change working directory

```
In [4]: os.chdir("E:/IMS")
```

```
In [5]: import csv  
        print("Reading csv -")  
        with open('sample.csv', newline='') as csvfile:  
            spamreader=csv.reader(csvfile, delimiter=' ', quotechar='|')  
            for row in spamreader:  
                print(', '.join(row))
```

“with open” keyword and csv.reader() function can be used to read file

```
Reading csv -  
I, am, learning, python
```



# Python Data Structure

- There are quite a few data structures available. The built-in data structures are: ***lists, tuples, dictionaries, strings, sets and frozensets.***
- Lists, strings and tuples are ordered sequences of objects.
- Unlike strings that contain only characters, list and tuples can contain any type of objects.
- Lists and tuples are like arrays. Tuples like strings are immutables. Lists are mutables so they can be extended or reduced at will.
- Sets are mutable unordered sequence of unique elements whereas frozensets are immutable sets.

# Object Declaration

- We may need to declare variables to store different type of values
- For declaration , there is no need to mention data type i.e. int, string or float
  - a = 123
  - b = "Python Data Structures"
  - c = 123.45
  - print("this is Int",a)
  - print("this is String",b)
  - print("this is Float",c)

```
In [2]: a = 123
...: b = "Python Data Structures"
...: c = 123.45
...:
...: print("this is Int",a)
...: print("this is String",b)
...: print("this is Float",c)
this is Int 123
this is String Python Data Structures
this is Float 123.45
```



- To know the type of a variable use “**type**” function
  - print("Type of a is",type(a))
  - print("Type of b is",type(b))
  - print("Type of c is",type(c))

```
In [3]: a=1.2  
        b="Test"  
        c=2  
        print("type of a is : ",(type(a)))  
        print("type of b is : ",(type(b)))  
        print("type of c is : ",(type(c)))
```

```
type of a is : <class 'float'>  
type of b is : <class 'str'>  
type of c is : <class 'int'>
```

To find the type of different variable, “type” function is used.

- In python, **lists** are part of the standard language.
- We could store multiple values in a single object. Lets say we would like to store a sequence of numbers/Id's, this could be achieved by lists
- List and Stacks, both are same in python

– >>> l = [1, 2, 3]

It is initiated with brackets []

– >>> l[0]

– 1

To access first element, we have to use bracket as index .

```
In [8]: temp = [1,2,3,4,5]
....: print("this is List", temp)
....:
....: # To print the ith element of list
....: print("1st and 2nd element of list",temp[0], temp[1])
this is List [1, 2, 3, 4, 5]
1st and 2nd element of list 1 2
```

- To insert an element at ith place
  - temp.insert(3,9)
  - print("List after insertion",temp)

```
In [9]: temp.insert(3,9)
...: print("List after insertion",temp)
List after insertion [1, 2, 3, 9, 4, 5]
```

- To append element to list we use append, this corresponds to push when we use list as stack
  - temp.append(10)
  - print("After append the list is",temp)

```
In [10]: temp.append(10)
...: print("After append the list is",temp)
After append the list is [1, 2, 3, 9, 4, 5, 10]
```

- To find index of any element
  - print(temp)
  - index = temp.index(5)
  - print("Index of 5 is ", index)

```
In [12]: print(temp)
...: index = temp.index(5)
...: print("Index of 5 is ", index)
...:
[1, 2, 3, 9, 4, 5, 10]
Index of 5 is 5
```

- To remove an element
  - print(temp)
  - temp.remove(5)
  - print("List after removing 5",temp)

```
In [13]: print(temp)
...: temp.remove(5)
...: print("List after removing 5",temp)
[1, 2, 3, 9, 4, 5, 10]
List after removing 5 [1, 2, 3, 9, 4, 10]
```

- To reverse the list

- print(temp)
- temp.reverse()
- print("List after reversing", temp)

```
In [14]: print(temp)
....: temp.reverse()
....: print("List after reversing", temp)
[1, 2, 3, 9, 4, 10]
List after reversing [10, 4, 9, 3, 2, 1]
```

- To sort the list

- print(temp)
- temp.sort()
- print("Sorted list",temp)

```
In [15]: print(temp)
....: temp.sort()
....: print("Sorted list",temp)
[10, 4, 9, 3, 2, 1]
Sorted list [1, 2, 3, 4, 9, 10]
```

- To remove last element of list or pop the element from stack

- print(temp)
- temp.pop()
- print("List after pop",temp)

```
In [16]: print(temp)
....: temp.pop()
....: print("List after pop",temp)
....:
[1, 2, 3, 4, 9, 10]
List after pop [1, 2, 3, 4, 9]
```

- Nested lists

- nested = [[1,2,3],[2,3,4],[5,6,7]]
- print("Nest list",nested)

```
In [17]: nested = [[1,2,3],[2,3,4],[5,6,7]]
....: print("Nest list",nested)
Nest list [[1, 2, 3], [2, 3, 4], [5, 6, 7]]
```

- To slice -> Slicing uses the symbol : to access to part of a list:

```
>>> list[first index:last index:step]
```

```
>>> list[:]
```

```
>>> a = [0, 1, 2, 3, 4, 5]
>>> print(a[2:])
>>> print(a[:2])
>>> print(a[2:-1])
```

```
[2, 3, 4, 5]
[0, 1]
[2, 3, 4]
```

Last element is accessed through -1

Lists is treated as stacks, that is a last-in, first-out data structures (LIFO).

An item can be added to a list by using the `append()` method.

The last item can be removed from the list by using the `pop()` method without passing any index to it.

```
In [6]: >>> stack = ['a','b','c','d']  
>>> stack.append('e')  
>>> stack.append('f')  
>>> stack
```

```
Out[6]: ['a', 'b', 'c', 'd', 'e', 'f']
```

```
In [8]: stack.pop()
```

```
Out[8]: 'f'
```

```
In [9]: stack
```

```
Out[9]: ['a', 'b', 'c', 'd', 'e']
```



# How to copy a list

- There are three ways to copy a list:

```
In [10]: a=[1,2,3,4]
         b=a[:]
         print(a)
         print(b)
```

```
[1, 2, 3, 4]
[1, 2, 3, 4]
```

```
In [13]: a=[1,6,9]
         import copy
         b=copy.copy(a)
         a.pop() ←
         print(a)
         print(b)
```

```
[1, 6]
[1, 6, 9]
```

Even the last item of “a” is popped out, the value of “b” which was copied earlier, remained intact.

# How to copy a list

- The preceding techniques for copying a list create shallow copies. It means that nested objects will not be copied. Consider this example:

```
In [1]: a = [1, 2, [3, 4]]  
       b = a[:]  
       a[2][0] = 10  
       a
```

```
Out[1]: [1, 2, [10, 4]]
```

```
In [2]: b
```

```
Out[2]: [1, 2, [10, 4]]
```

```
In [4]: import copy  
       a = [1, 2, [3, 4]]  
       b = copy.deepcopy(a)  
       a[2][0] = 10  
       print("a : ",a)  
       print("b : ",b)
```

```
a : [1, 2, [10, 4]]  
b : [1, 2, [3, 4]]
```

Here, the value of “a” is changed after copying. But the value of “b” also changed correspondingly

So, to counter such changes, “deepcopy” function is used

- The other main data type is the dictionary. The dictionary allows you to associate one piece of data (a "key") with another (a "value"). The analogy comes from real-life dictionaries, where we associate a word (the "key") with its meaning. It's a little harder to understand than a list, but Python makes them very easy to deal with.
- It is best to think of a dictionary as an unordered set of key: value pairs, with the requirement that the keys are unique (within one dictionary).
- A pair of braces creates an empty dictionary: {}
- Placing a comma-separated list of key : value pairs within the braces adds initial key : value pairs to the dictionary; this is also the way dictionaries are written on output.
- Declaration of dictionary
  - Lets say we would like to create an object which stores the name and scores of students. This could be achieved with dictionary in **keys : values** format
- score = {'rob' : 85, 'kevin' : 70}
- print("The dictionary is",score)

```
In [18]: score = {'rob' : 85, 'kevin' : 70}
...: print("The dictionary is",score)
The dictionary is {'rob': 85, 'kevin': 70}
```

- To get the score of any student or value in dictionary
  - `print(score)`
  - `mar = score['kevin']`
  - `print("Marks of Kevin are",mar)`
  - `print("Marks of rob are",score['rob'])`    *# print the score of rob*

```
In [20]: print(score)
...: mar = score['kevin']
...: print("Marks of Kevin are",mar)
...: print("Marks of rob are",score['rob'])      # print the score of rob
...:
{'rob': 85, 'kevin': 70}
Marks of Kevin are 70
Marks of rob are 85
```

- Add new elements or students to the dictionary
  - print(score)
  - score['jim']= 92
  - score['tomy']= 56
  - print("The dictionary after adding jim and tomy -",score)

```
In [21]: print(score)
...: score['jim']= 92
...: score['tomy']= 56
...: print("The dictionary after adding jim and tomy -",score)
{'rob': 85, 'kevin': 70}
The dictionary after adding jim and tomy - {'rob': 85, 'kevin': 70, 'jim': 92, 'tomy': 56}
```

- “keys” gives you keys in dictionary, i.e. names of all students in this case
  - print(score)
  - names = list(score.keys())
  - print("List of students -",names)

```
In [22]: print(score)
...: names = list(score.keys())
...: print("List of students -",names)
{'rob': 85, 'kevin': 70, 'jim': 92, 'tomy': 56}
List of students - ['rob', 'kevin', 'jim', 'tomy']
```

- “values” gives you values in dictionary, i.e. marks of all students in this case
  - print(score)
  - marks = list(score.values())
  - print("List of marks -",marks)

```
In [23]: print(score)
...: marks = list(score.values())
...: print("List of marks -",marks)
{'rob': 85, 'kevin': 70, 'jim': 92, 'tomy': 56}
List of marks - [85, 70, 92, 56]
```

- We can also look at the marks in sorted order i.e. non descending order
  - `print(score)`
  - `sortedMarks = sorted(score.values())`
  - `print("List of marks in sorted order -",sortedMarks)`

```
In [24]: print(score)
...: sortedMarks = sorted(score.values())
...: print("List of marks in sorted order -",sortedMarks)
{'rob': 85, 'kevin': 70, 'jim': 92, 'tomy': 56}
List of marks in sorted order - [56, 70, 85, 92]
```



- Checks if a particular student name is there in the dictionary
  - print(score)
  - check = 'kory' in score
  - print("If kory is there in dictionary -",check)

```
In [25]: print(score)
...: check = 'kory' in score
...: print("If kory is there in dictionary -",check)
{'rob': 85, 'kevin': 70, 'jim': 92, 'tomy': 56}
If kory is there in dictionary - False
```

```
check2 = 'jim' in score
print("If jim is there in dictionary -",check2)
```

```
In [26]: check2 = 'jim' in score
...: print("If jim is there in dictionary -",check2)
If jim is there in dictionary - True
```

In addition to keys and values methods, there is also the items method that returns a list of items of the form (key, value). The items are not returned in any particular order.

In order to get the value corresponding to a specific key, use get or pop :

```
In [25]: d = {'first':'string value', 'second':[1,2]}  
         d.items()
```

```
Out[25]: dict_items([('first', 'string value'), ('second', [1, 2])])
```

```
In [21]: d.get('first')
```

```
Out[21]: 'string value'
```

```
In [22]: d.pop('first')  
         d
```

```
Out[22]: {'second': [1, 2]}
```

The difference between get and pop is that pop also removes the corresponding item from the dictionary

**popitem** removes and returns a pair (key, value); you do not choose which one because a dictionary is not sorted :

```
In [26]: d.popitem()
```

```
Out[26]: ('first', 'string value')
```

```
In [27]: d
```

```
Out[27]: {'second': [1, 2]}
```

To create a new object, use the **copy** method (shallow copy):

-> d2 = d1.copy()

You can **clear** a dictionary (i.e., remove all its items) using the clear() method:

-> d2.clear()

The clear() method **deletes** all items whereas **del()** deletes just one:

```
In [6]: d = {'a':1, 'b':2, 'c':3}
        del d['a']
        d.clear()
        d
Out[6]: {}
```

All the values of “d” is cleared by the function “clear”

- A queue is a first in, first out (FIFO) structure. This means that the first item to join the queue is the first to leave the queue.
- Importing & declaring a queue
  - from collections import deque
  - queue = deque(["aa", "bb", "cc"])
  - print(queue)

```
In [27]: from collections import deque
...: queue = deque(["aa", "bb", "cc"])
...: print(queue)
...:
deque(['aa', 'bb', 'cc'])
```

- Appending element to queue
  - queue.append("dd")
  - queue.append("ee")
  - print("After appending dd and ee, queue is",queue)

```
In [28]: queue.append("dd")
...: queue.append("ee")
...: print("After appending dd and ee, queue is",queue)
After appending dd and ee, queue is deque(['aa', 'bb', 'cc', 'dd', 'ee'])
```

- Removing left element from queue
  - `left = queue.popleft()`                      *# The first to arrive now leaves*
  - `print("After removing left element, queue is",queue)`
  - `secLeft = queue.popleft()`                      *# The second to arrive now leaves*
  - `print("After removing left element, queue is",queue)`

```
In [29]: left = queue.popleft()           # The first to arrive now leaves
...: print("After removing left element, queue is",queue)
...: secLeft = queue.popleft()           # The second to arrive now leaves
...: print("After removing left element, queue is",queue)
...:
```

After removing left element, queue is deque(['bb', 'cc', 'dd', 'ee'])

After removing left element, queue is deque(['cc', 'dd', 'ee'])

In Python, tuples are part of the standard language. This is a data structure very similar to the list data structure. The main difference being that tuple manipulation are faster than list because tuples are immutable.

To create a tuple, place values within brackets:

```
In [12]: l = (1, 2, 3)
print("first element of tuple l : ",l[0])
m=4,5,6 ←
print("second element of tuple m : ",m[1])

first element of tuple l : 1
second element of tuple m : 5
```

It is also possible to create a tuple without parentheses, by using commas:

To create a tuple with a single element, you must use the comma:

```
>>> singleton = (1, )
```



```
In [15]: t=(1,) #to create a singleton tuple "," should be used
print("Singleton tuple",t)
r=t*5#repeat a tuples by multiplying a tuple by a number:
print("repeated tuple",r)
r += (9,)#you can concatenate tuples and use augmented assignment (*=, +=):
print("concatenated tuple",r)
```

Singleton tuple (1,)

repeated tuple (1, 1, 1, 1, 1)

concatenated tuple (1, 1, 1, 1, 1, 9)

Since the tuple is multiplied by 5, it has grown into size 5 tuple from singleton tuple

9 as value of singleton tuple , is added at the end of tuple

Tuples are useful because there are

- faster than lists
- protect the data, which is immutable
- tuples can be used as keys on dictionaries

In addition, it can be used in different useful ways:

## 1. Tuples as key/value pairs to build dictionaries

```
In [16]: d = dict([('jan', 1), ('feb', 2), ('march', 3)])  
          d['feb']
```

```
Out[16]: 2
```

```
In [18]: (x,y,z) = ('a','b','c')  
          x
```

```
Out[18]: 'a'
```

```
In [19]: (x,y,z) = range(3)  
          z
```

```
Out[19]: 2
```

Dictionary is created using tuples

Multiple values can be assigned using tuples

“range” function generates sequence of numbers. E.g. range(3) = (0,1,2)

## Tuple Unpacking

Tuple unpacking allows to extract tuple elements automatically if the list of variables on the left has the same number of elements as the length of the tuple

```
In [20]: data = (1,2,3)
         (p,q,r) = data
         p
```

```
Out[20]: 1
```

```
In [21]: (p,q)=(q,p)
         print("swapped p : ",p)
         swapped p : 2
```

Tuple can be use as **swap function**  
This code reverses the contents of 2 variables p and q:

## Length

To find the length of a tuple, you can use the len() function:

```
>>> t= (1,2,3,4,5)
>>> len(t)
5
```

## Slicing (extracting a segment)

```
>>> t = (1,2,3,4,5)
>>> t[2:]
(3, 4, 5)
```

## Copy a tuple

To copy a tuple,:

```
>>> t = (1, 2, 3, 4, 5)
>>> newt = t
>>> t[0] = 5
>>> newt
(1, 2, 3, 4, 5)
```

```
In [22]: t=(1,2,3,4,5)
print("Length of tuple : ",len(t))
s=t[2:]
print("Slice of tuple : ",s)
copyt = t
t[0]=5
print("Original tuple : ",t)
print("Copy of tuple : ",copyt)
```

```
Length of tuple : 5
Slice of tuple : (3, 4, 5)
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-22-9ac0407aa6c6> in <module>()
      4 print("Slice of tuple : ",s)
      5 copyt = t
----> 6 t[0]=5
      7 print("Original tuple : ",t)
      8 print("Copy of tuple : ",copyt)
```

```
TypeError: 'tuple' object does not support item assignment
```

Tuple is immutable. So, we can't assign values to it.

Tuple are **not fully immutable !!**

If a value within a tuple is mutable, then you can change it:

```
>>> t = (1, 2, [3, 10])
```

```
>>> t[2][0] = 9
```

```
>>> t
```

```
(1, 2, [9, 10])
```

## Math and comparison

comparison operators and mathematical functions can be used on tuples. Here are some examples:

```
>>> t = (1, 2, 3)
```

```
>>> max(t)
```

```
3
```

## Convert a tuple to a string

You can convert a tuple to a string with either:

```
>>> str(t)
```

```
or
```

```
>>> `t`
```

```
In [23]: t = (1, 2, [3, 10])  
         t[2][0]=4  
         t
```

```
Out[23]: (1, 2, [4, 10])
```

```
In [26]: t=1,2,3,6  
         max(t)
```

```
Out[26]: 6
```

```
In [27]: str(t)
```

```
Out[27]: '(1, 2, 3, 6)'
```

- Sets are constructed from a sequence (or some other iterable object). Since sets cannot have duplicated, there are usually used to build sequence of unique items (e.g., set of identifiers).

```
In [8]: animals = {'dog', 'cat', 'lion', 'tiger'}  
print("first input : ", animals)  
animals1 = {'dog', 'dog', 'cat', 'lion', 'tiger'}  
print("second input : ", animals1)
```

```
first input : {'lion', 'dog', 'cat', 'tiger'}  
second input : {'lion', 'dog', 'cat', 'tiger'}
```

“dog” is repeated two times,  
but sets cannot keep  
duplicate item, so one “dog”  
is removed

- Sets are mutable unordered sequence of unique elements
- Sets are constructed from a sequence (or some other iterable object)
- Since sets cannot have duplicated, there are usually used to build sequence of unique items

- Quick look

```
In [9]: a = set([1, 2, 3, 4])  
b = set([3, 4, 5, 6])  
print("OR function in set : ",a | b)  
print("AND function in set : ",a & b)  
print("Subset function in set : ",a < b)  
print("Symmetric difference function in set : ",a ^ b)
```

OR function in set : {1, 2, 3, 4, 5, 6}

AND function in set : {3, 4}

Subset function in set : False

Symmetric difference function in set : {1, 2, 5, 6}

“OR” combines all the element and takes unique value of combined element

“AND” picks all the elements which are Common in both the sets

“<” function looks, if all the elements of “a” is present in “b”

“^” function combines all the element and removes all the common element

- Sets are iterable
  - `i = 'cat' in animals`
  - `print(i)`
  - `j = 'crow' in animals`
  - `print(j)`

```
In [35]: i = 'cat' in animals  
...: print(i)  
...:  
...: j = 'crow' in animals  
...: print(j)  
...:
```

```
True  
False
```

animals contain( lion, cat, dog and tiger. Since “cat” is in “animals” set, the statement is true. But crow is not part of “animals” set, so output is false.



- Set always give non-duplicate set of elements

- a = set('abracadabra')
- b = set('alacazam')
  
- print("Set a is",a)
- print("Set b is",b)

```
In [36]: a = set('abracadabra')
...: b = set('alacazam')
...:
...: print("Set a is",a)
...: print("Set b is",b)
Set a is {'r', 'b', 'a', 'd', 'c'}
Set b is {'l', 'a', 'm', 'z', 'c'}
```

- Letters in a but not in b
  - $c = a - b$
  - `print("letters in a but not in b",c)`

```
In [38]: c = a - b
...: print("letters in a but not in b",c)
letters in a but not in b {'b', 'd', 'r'}
```

- Letters in either a or b
  - $d = a \cup b$
  - `print("letters in either a or b",d)`

```
In [39]: d = a | b
...: print("letters in either a or b",d)
letters in either a or b {'l', 'r', 'b', 'a', 'd', 'm', 'z', 'c'}
```

- Letters in both a and b
  - $e = a \& b$
  - `print("letters in both a and b",e)`

```
In [40]: e = a & b
...: print("letters in both a and b",e)
...:
letters in both a and b {'c', 'a'}
```

- Letters in a or b but not both
  - $f = a \wedge b$
  - `print("letters in a or b but not both",f)`

```
In [41]: f = a ^ b
...: print("letters in a or b but not both",f)
...:
letters in a or b but not both {'l', 'r', 'b', 'm', 'd', 'z'}
```

- Sets are mutable, and may therefore not be used, for example, as keys in dictionaries.
- Another problem is that sets themselves may only contain immutable (hashable) values, and thus may not contain other sets.
- Because sets of sets often occur in practice, there is the **frozenset** type, which represents immutable (and, therefore, hashable) sets.

```
In [31]: a = frozenset([1, 2, 3])  
b = frozenset([2, 3, 4])  
c = a.union(b)  
print(c)
```

```
frozenset({1, 2, 3, 4})
```

```
In [32]: c[3] = 5
```

```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-32-3daaaf9dc7c2> in <module>()  
----> 1 c[3] = 5
```

```
TypeError: 'frozenset' object does not support item assignment
```

Frozenset is immutable. So, we can't assign values to it.

## **Frozensets : Using set as key in a dictionary**

If you want to use a set as a key dictionary, you will need frozenset:

```
In [38]: fa = {frozenset([1,2]): 1}
          fa[ frozenset([1,2]) ]<
Out[38]: 1
```

1 as key value, has been set to fa (frozenset). It can be retrieved.

## **Frozensets : Methods**

frozensets have less methods than sets.

There are some operators similar to sets (intersection(), union(), symmetric\_difference(), difference(), issubset(), isdisjoint(), issuperset()) and a copy() method.



# **Thank You.**