One Step In Changing Education Chain...

RED & WHITE®

GROUP OF INSTITUTES

STUDY MATERIAL

CORE PYTHON

Table of Contents

#1 INTRODUCTION TO PYTHON

WHAT IS PYTHON LANGUAGE?

1. પાયથન એક પ્રોગ્રામિંગ language છે જે web-development, software development, mathematics અને system scripting માટે વપરાય છે.
2. પાયથન english ભાષા જેવી સરળ syntax ધરાવે છે જેથી આ language શીખવામાં ખુબ જ સરળ પડે છે.
3. પાયથનની સરળ sytax ને કારણે developers બીજી કોઈ પ્રોગ્રામિંગ language ની સરખામણી માં ઘણી ઓછી lines માં code લખી શકે છે.
4. પાયથન interpreter system પર રન થાય છે, એટલે કે જેમ જેમ code લખાતો જાય છે તેમ તરત જ તે code રન પણ થતો જાય છે. આ રીતે prototyping ખુબ જ ઝડપથી થઇ શકે છે.
5. પાયથનને procedural અને object -oriented તેમ બંને રીતે ઉપયોગ માં લઇ શકાય છે.
6. પાયથન scope બનાવવા માટે indentation (white space) પર આધાર રાખે છે, જેવા કે loops, functions અને classes ના scope. બાકીની કોઈ પ્રોગ્રામિંગ language આ હેતુ માટે curly-brackets { } નો ઉપયોગ કરે છે.

HISTORY OF PYTHON

1. પાયથન નું implementation **CWI, Netherland December 1989** માં **Guido Van Rossum** દ્વારા શરૂ કરવામાં આવ્યું હતું.

GUIDO VAN ROSSUM

2. February 1991 માં Guido Van Rossum એ પાયથન નો પહેલો code versin 0.9.0 publish કરેલો.
3. પાયથન language એ ABC પ્રોગ્રામિંગ language પરથી બનેલી language chhe જે પોતે Exception handling અને Amoeba operating system સાથે capable હતી.
4. પાયથન મૂળ 2 પ્રોગ્રામિંગ languages પરથી influenced થયેલ છે:
   A. ABC language.
   B. Modula-3
5. 1994 માં પાયથન version 1.0 ને રિલીઝ કરવામાં આવ્યું હતું.
6. હાલ માં પાયથન નું latest version 3.8 ચાલે છે.

PYTHON VERSIONS

1. પાયથન ના મૂળ 2 versions main ગણાય છે:

   Python 2.x અને **Python 3.x**

   x = 1,2,3,...
2. ઘણા સમયથી ચાલી રહેલા Python 2.x ની lifecycle નો end 2020 માં already થઇ ચુક્યો છે. એટલે કે હવે થી ક્યારેય Python 2.x વિષે કોઈ પણ પ્રકારનું update કે maintanence સપોર્ટ કરવામાં આવશે નહિ.
3. Python 3.x એ હાલમાં પાયથન નું latest અને fast version છે.

Python Version	Release Date
Python 1.0	January 1994
Python 1.6	September 5, 2000
Python 2.0	October 16, 2000
Python 2.7	July 3, 2010
Python 3.0	December 3, 2008
Python 3.8	October 14, 2019

INSTALLATION OF PYTHON

A. INSTALLATION PROCESS ON MACOS

STEP - 1 First go to this link: https://www.python.org/

STEP - 2

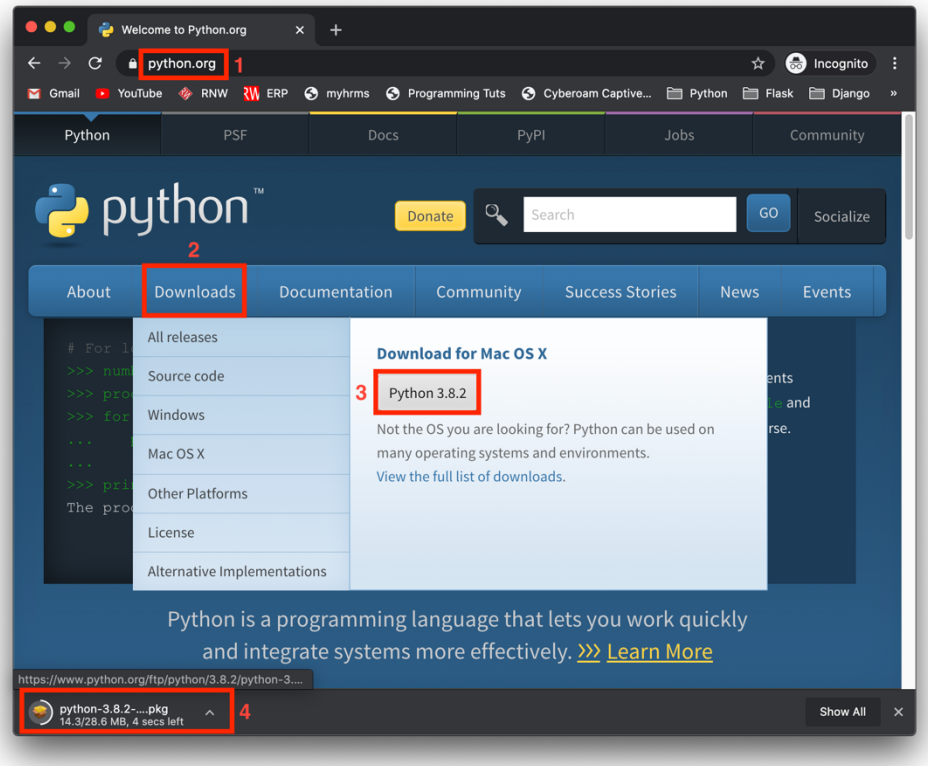
Hover on **Downloads** button from navigation bar.

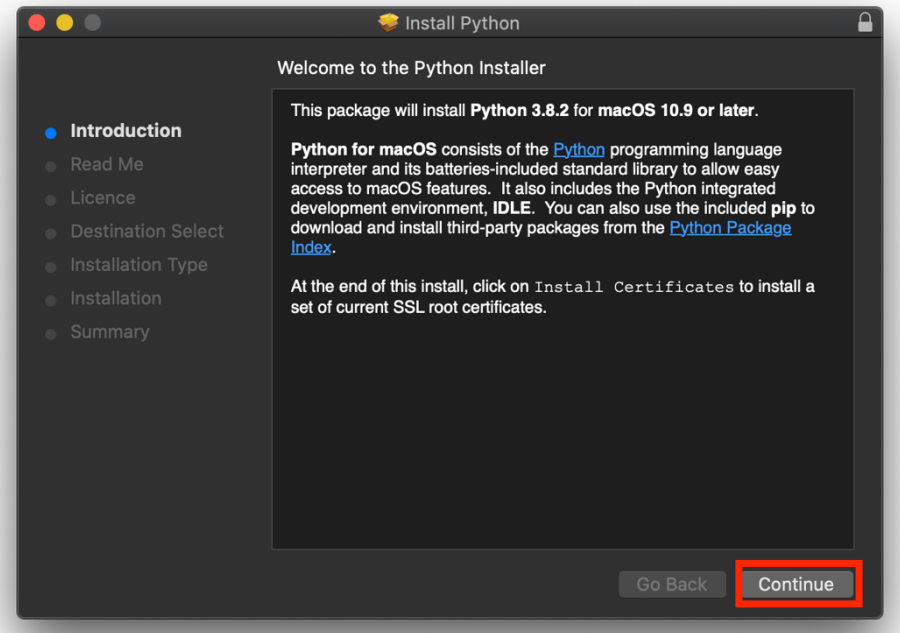
STEP - 3

Click on **Python 3.8.2** button.

STEP - 4

Download will start.



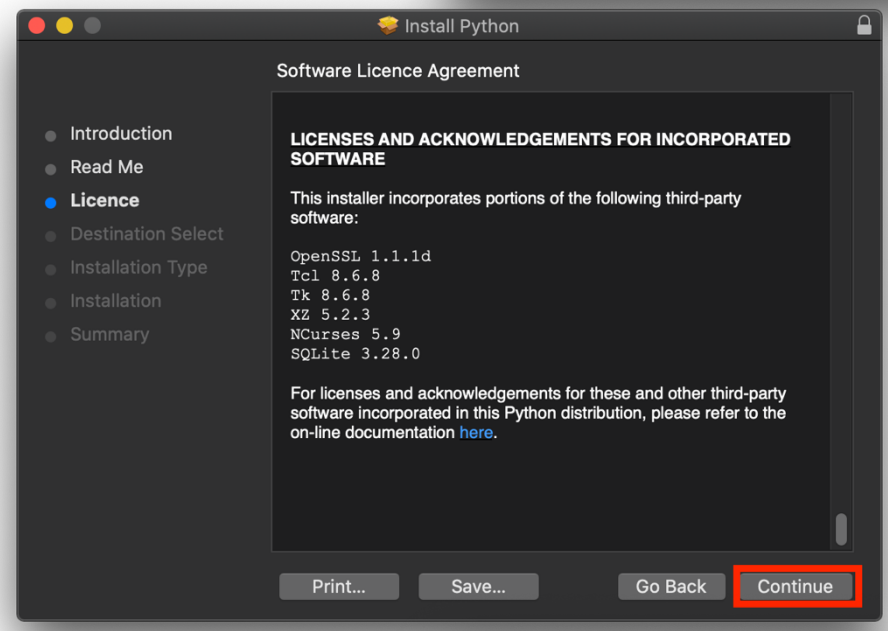


STEP - 5

Now open downloaded file and click on **Continue** button.

STEP - 6

Click on **Continue** button again.

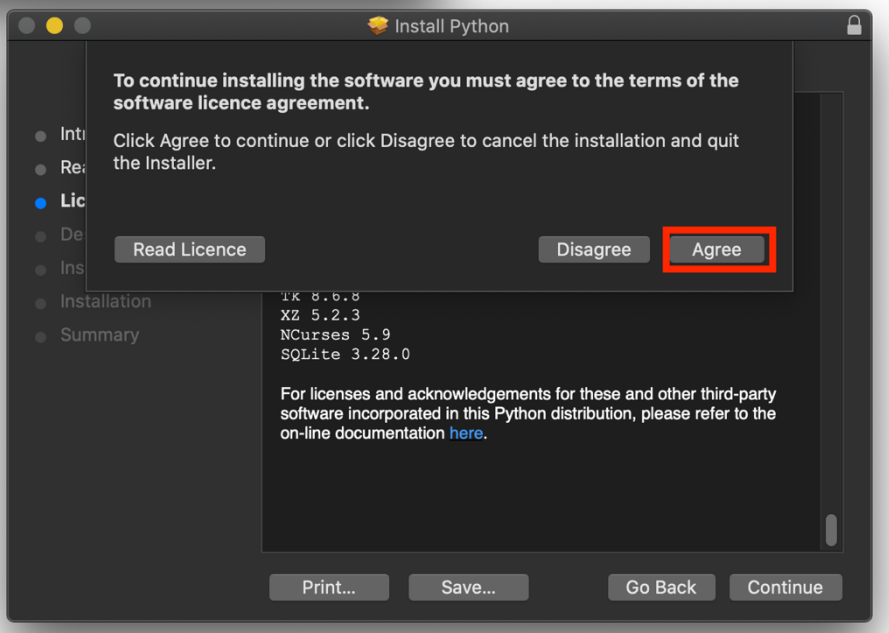


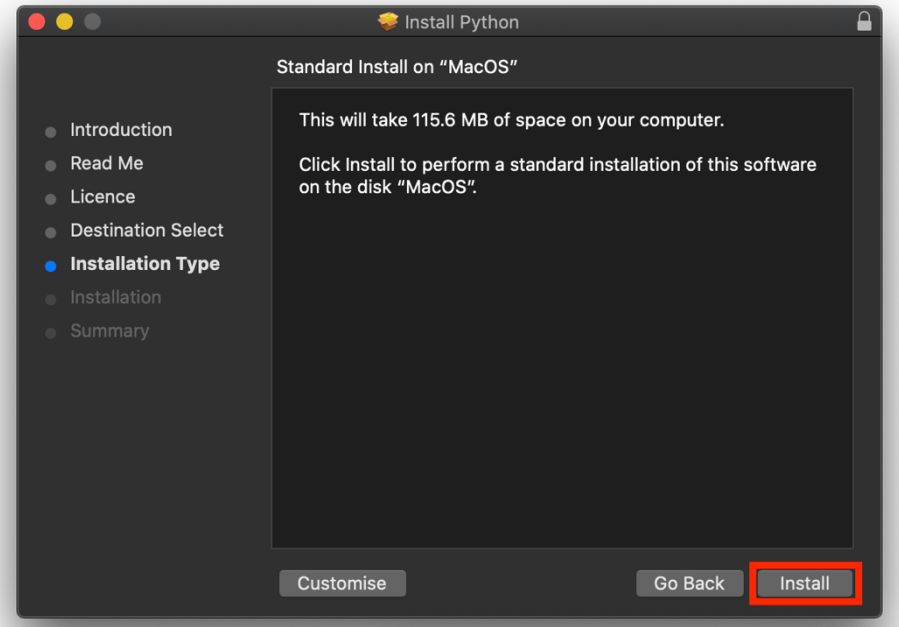
STEP - 7

After reading all licence agreement, click on **Continue** button.

STEP - 8

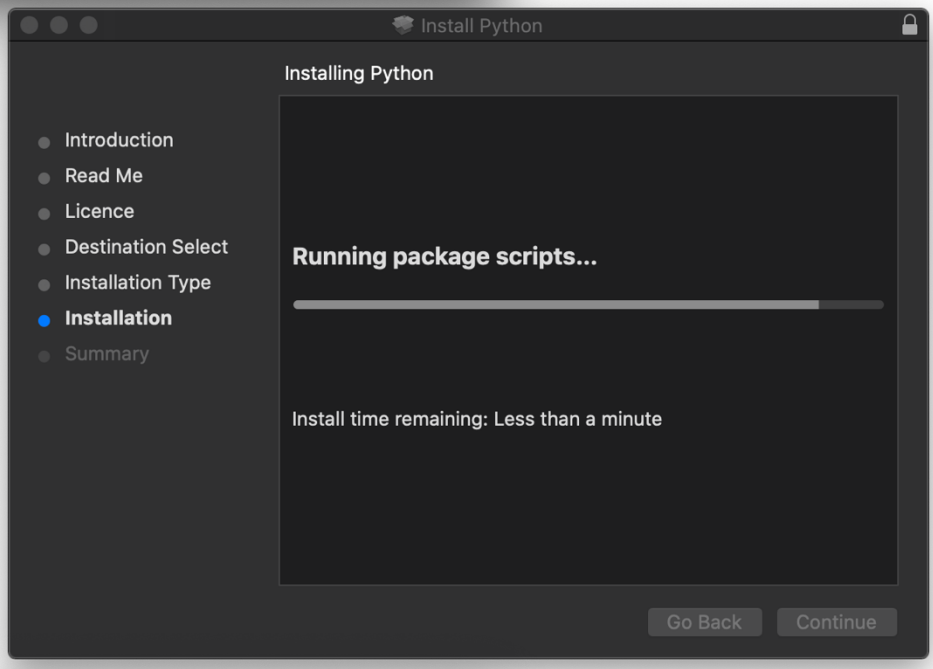
Click on **Agree** button.





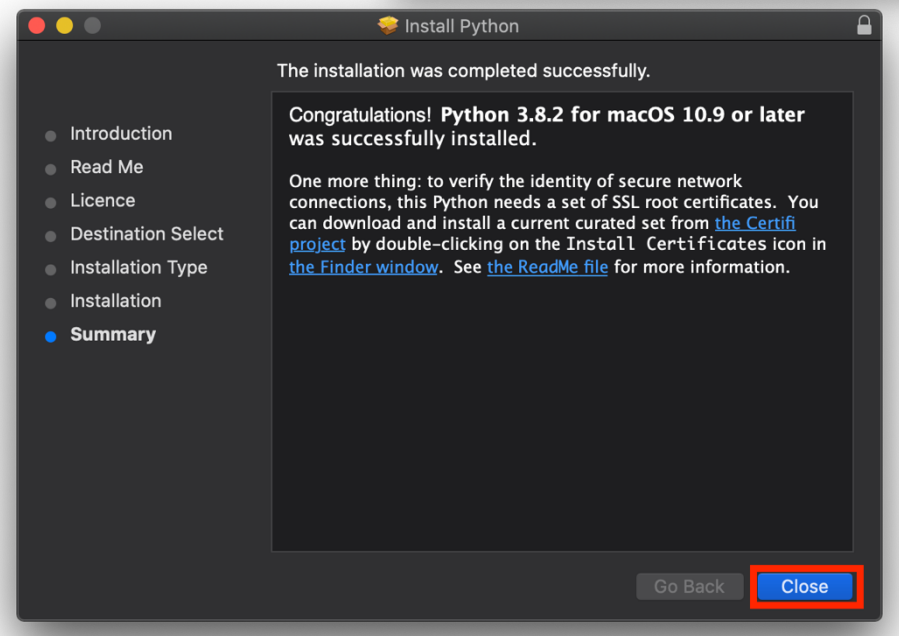
STEP - 9

Click **Install** button.



STEP - 10

Setup is now installing.



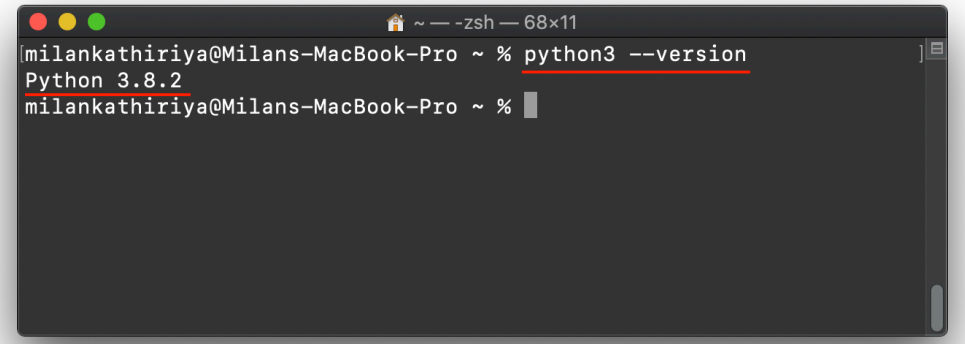
STEP - 11

Now click on **Close** button.
Python installation successfully done.

STEP - 12

Now open your **Terminal** and write command **python3 –version**

Press Enter and that shows up installed version of Python.



B. INSTALLATION PROCESS ON WINDOWS

STEP - 1 First go to this link: https://www.python.org/

STEP - 2

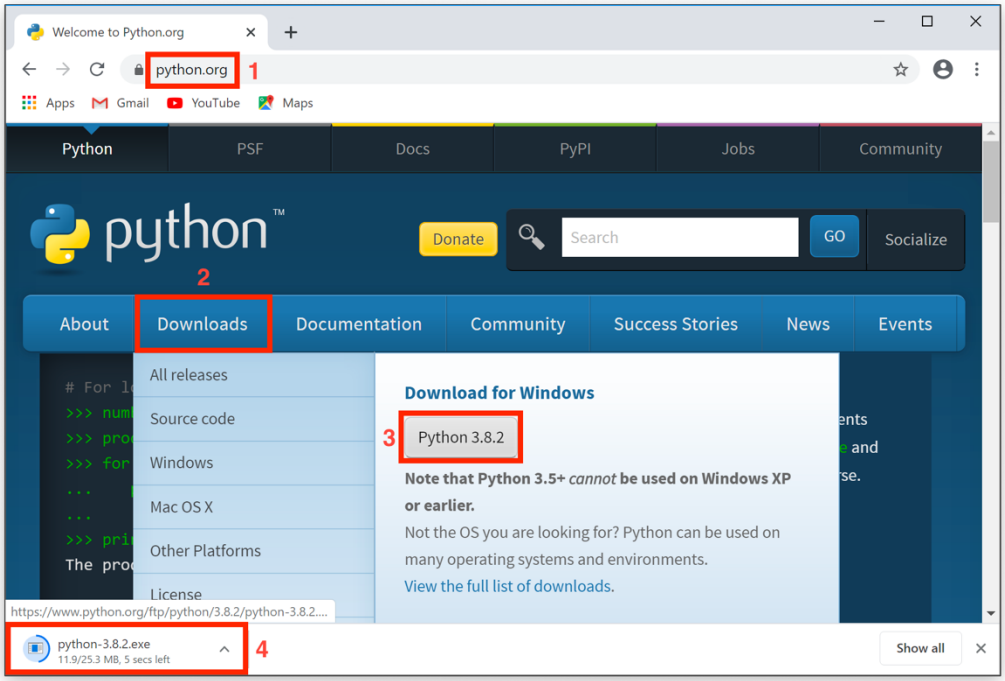
Hover on **Downloads** button from navigation bar.

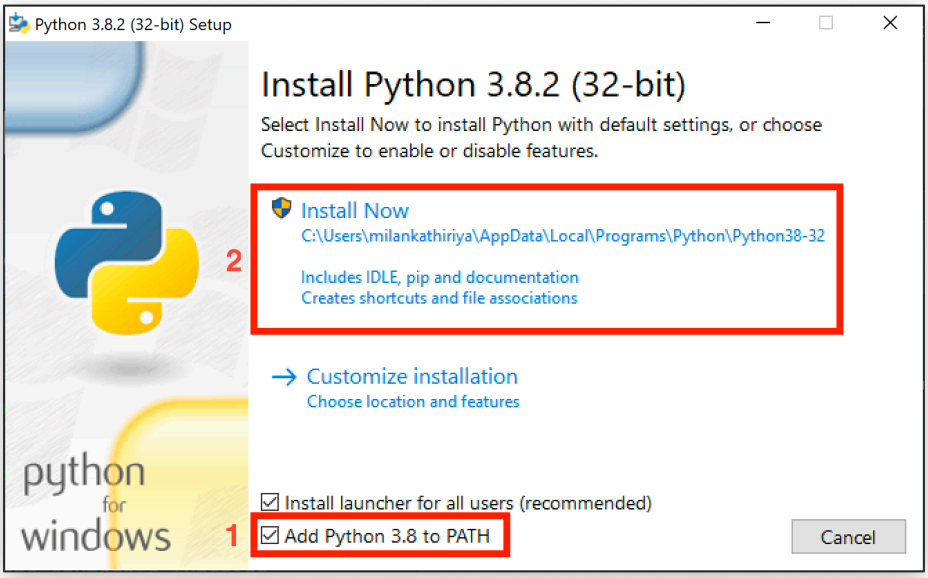
STEP - 3

Click on **Python 3.8.2** button.

STEP - 4

Download will start.





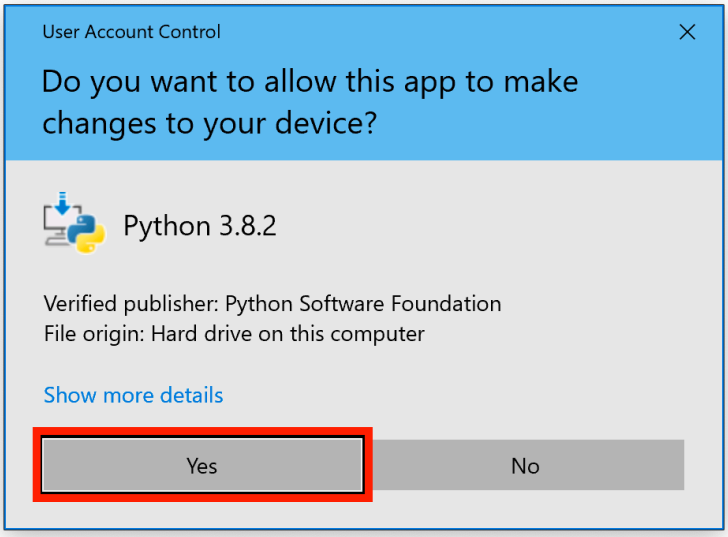
STEP - 5

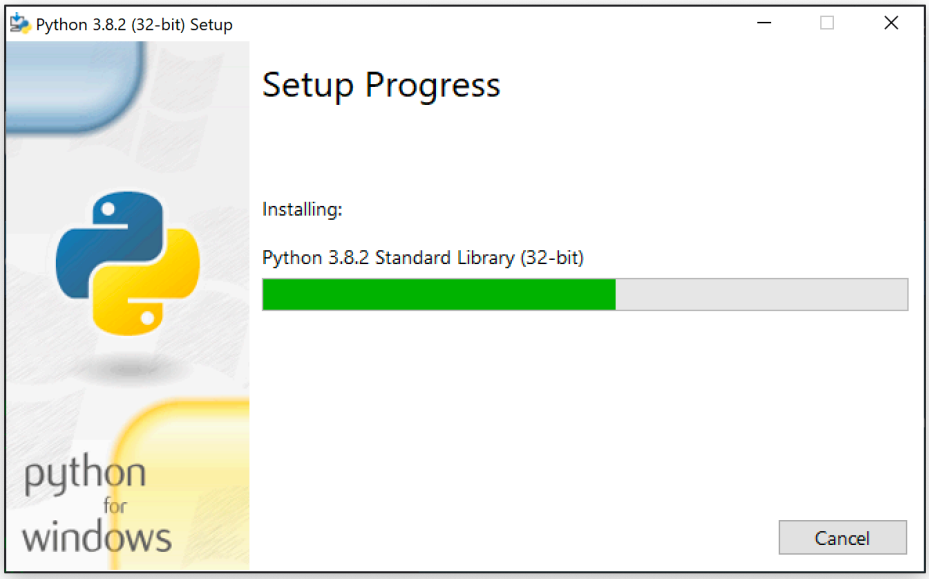
First check **Add Python 3.8.2 to PATH**

Then, Click on **Install Now**

STEP - 6

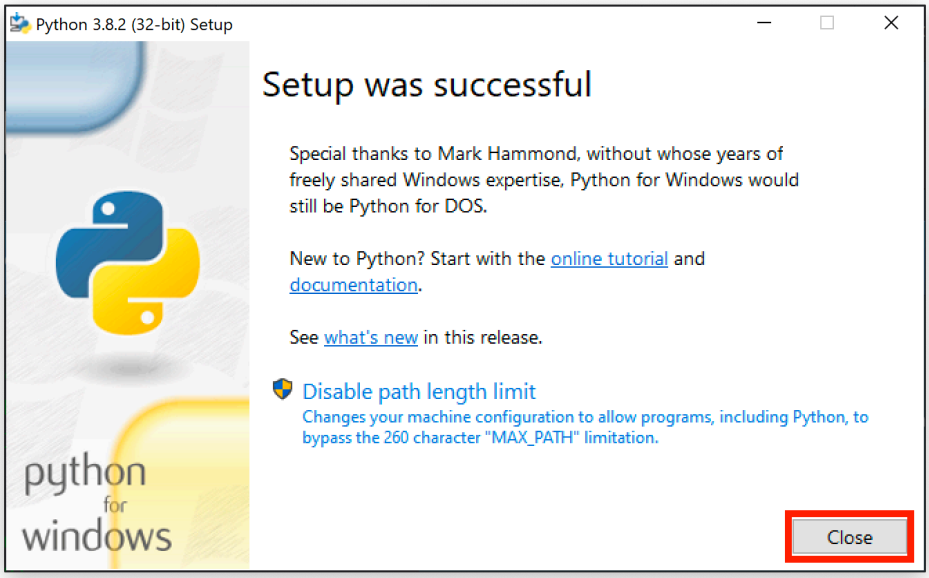
Click on **Yes** button.





STEP - 7

Setup is now installing.



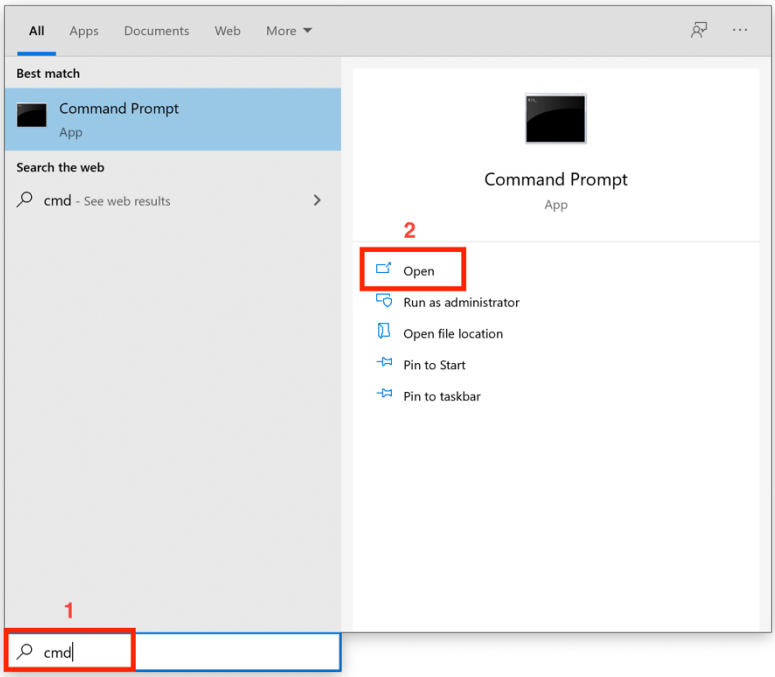
STEP - 8

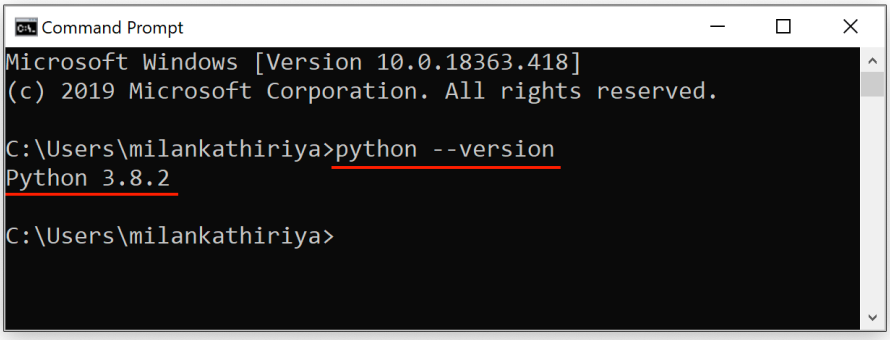
Now, click on **close** button.

Python installation successfully done.

STEP - 9

Search **cmd** in search bar and click **Open**.





STEP - 10

Now open your **cmd** and write command
python –version

Press Enter and that shows up installed version of Python.

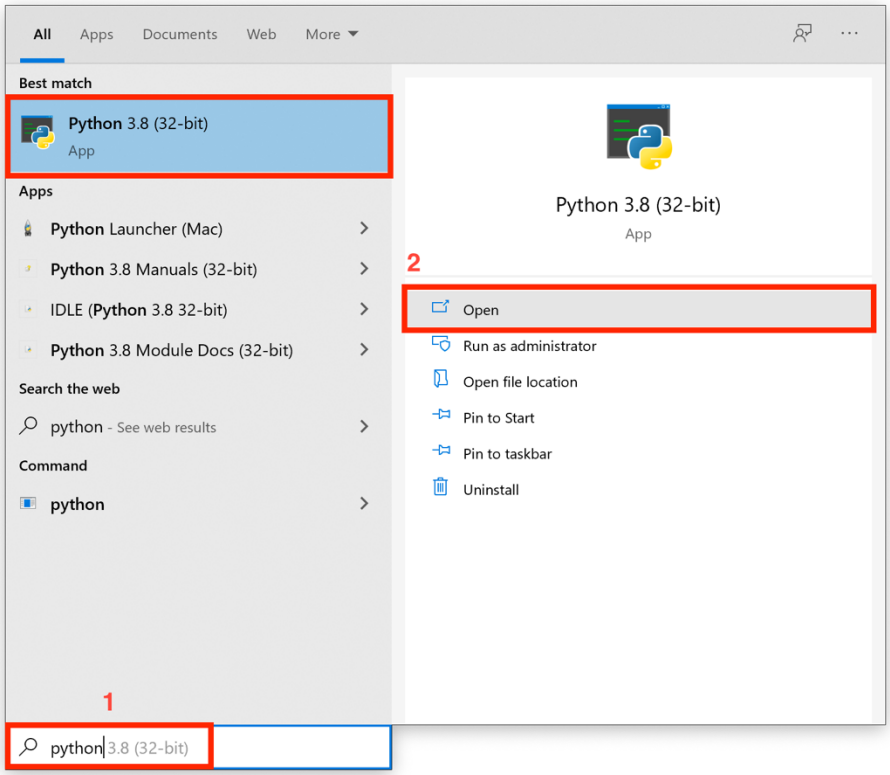
PYTHON SHELL & IDLE

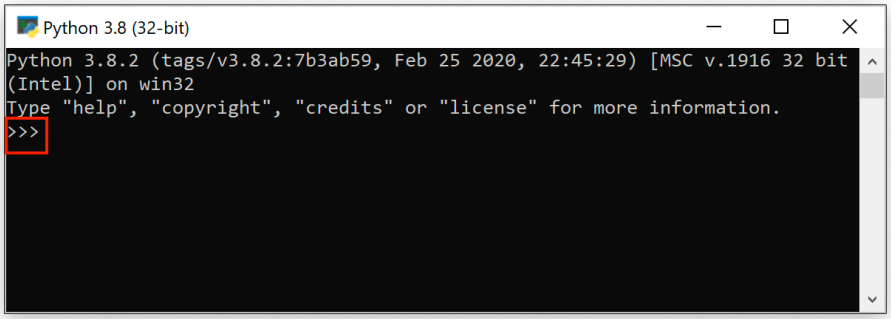
1. પાયથન આપણને **Shell** અને **IDLE (Integrated Development Environment)** જેવા built-in platform પ્રદાન કરે છે જેમાં આપણે પાયથન નું coding કરી શકીએ છીએ.
2. પાયથન ને install કરતાની સાથે જ shell અને IDLE પણ built-in સાથે જ install થઇ જાય છે.

A. Opening SHELL

STEP - 1

Search **python** in search bar and click on **Open** to open Python Shell.





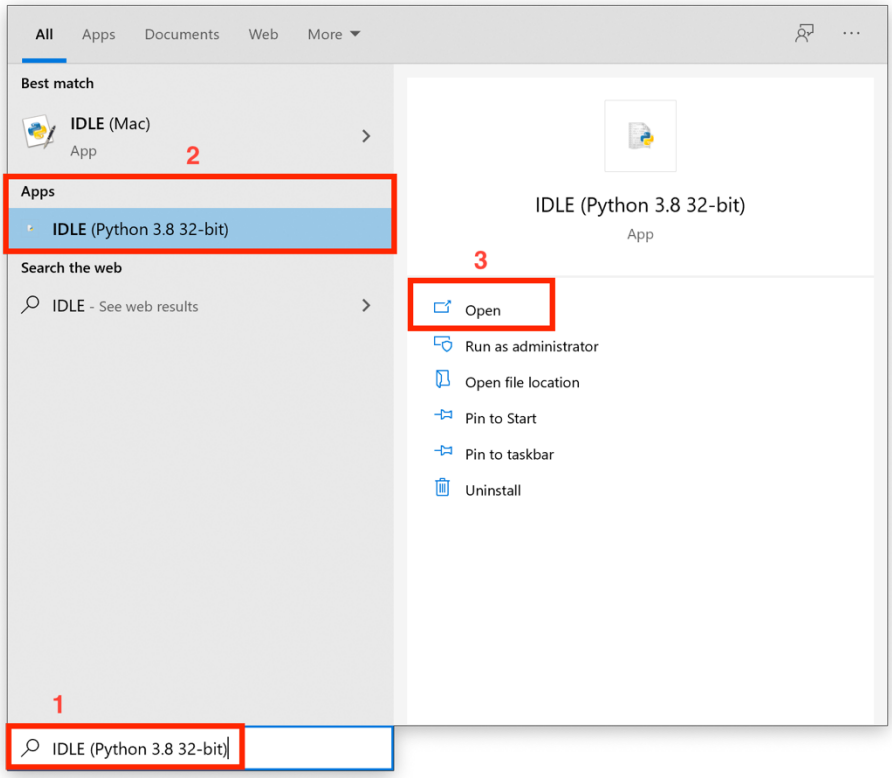
STEP - 2

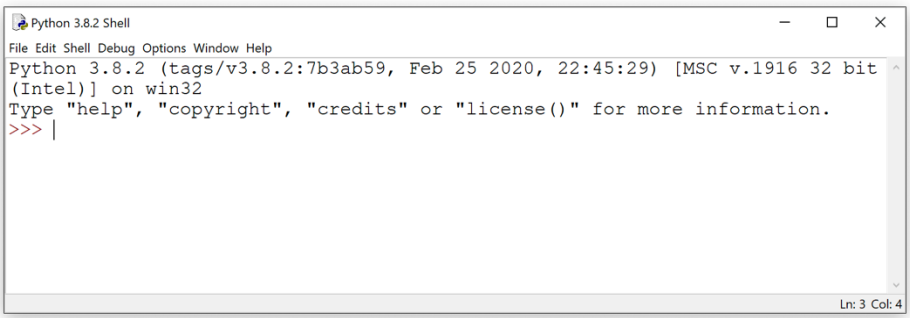
In shell window, **>>>** indicates interactive **interpreter** mode.

B. Opening IDLE

STEP - 1

Search **IDLE** in search bar and click on **Open** to open Python IDLE.





STEP - 2

In IDLE window, **>>>** indicates interactive **interpreter** mode.

- IDLE ને open કરતા જ તે by -default interactive **interpreter** mode માં open થાય છે. પણ IDLE ને **script** mode માં પણ ઉપયોગ માં લઇ શકાય છે.

FIRST PYTHON PROGRAM / TYPES OF MODE TO RUN PROGRAM

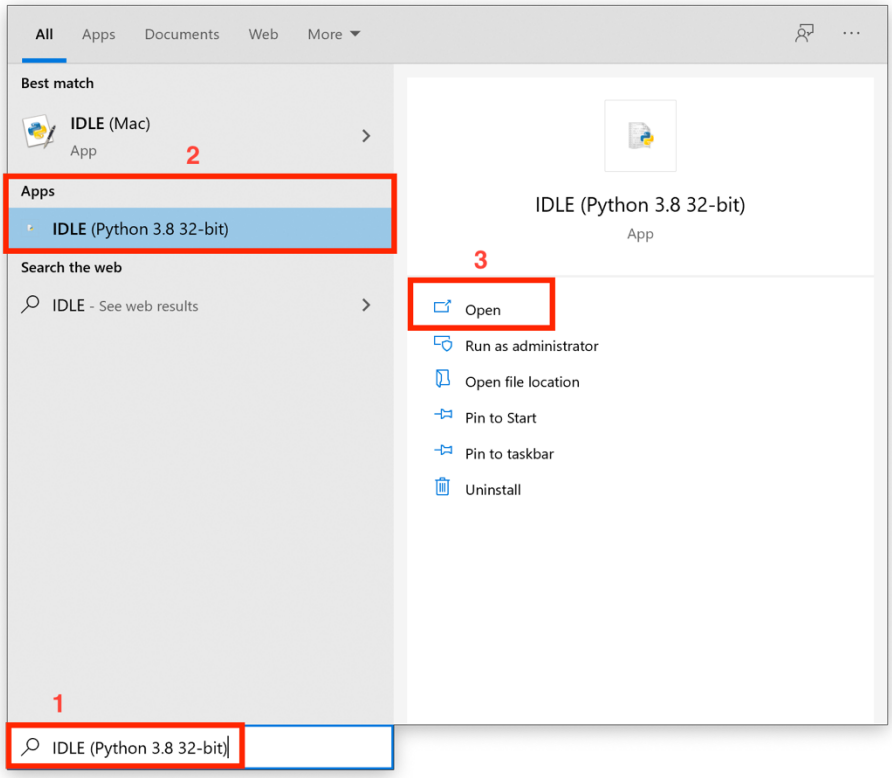
1. પાયથન પ્રોગ્રામને રન કરવા માટે 2 મેથડ છે:
   - **A.** Using Interpreter Mode
   - **B.** Using Script Mode

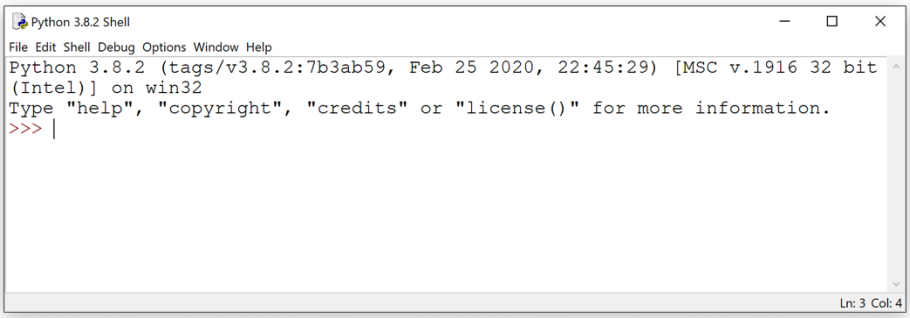
A. Using Interpreter Mode

- **Interpreter mode** એટલે કે પ્રોગ્રામ **line-by-line** રન થવો.
- એટલે કે જેમ જેમ આપડે એક એક લાઈન અખત જઈસુ તેમ તેમ તે લાઈન રન થતી રહેશે.
- જેવું એક લાઈન નું coding કરીને બીજી લાઈન માં જવા ગયા કે તરત જ તે પેલી લાઈન નું output આપણને જોવા મળશે.
- Interpreter mode માં આપડે આખો પ્રોગ્રામ એકસાથે લખીને પાછળ થી રન કરી શકતા નથી.
- આ mode ફક્ત **testing** અથવા તો **learning** માટે વપરાતો હોય છે.
- Interpreter mode એ પાયથન shell અને IDLE બંને પ્રદાન કરે છે.
- આપડે **IDLE** ની મદદથી interpreter mode ને સમજીશું.

STEP - 1

Search **IDLE** in search bar and click on **Open** to open Python IDLE.





STEP - 2

In IDLE window, **>>>** indicates interactive **interpreter** mode.

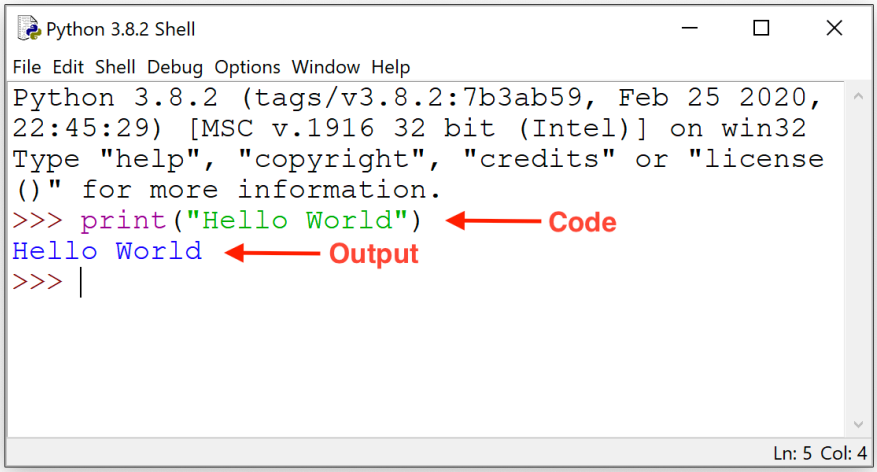
STEP - 3

To print **Hello World**, write below code:

print("Hello World")

Press **Enter** for see output.

Notice here, Python doesn't have **semicolon**.



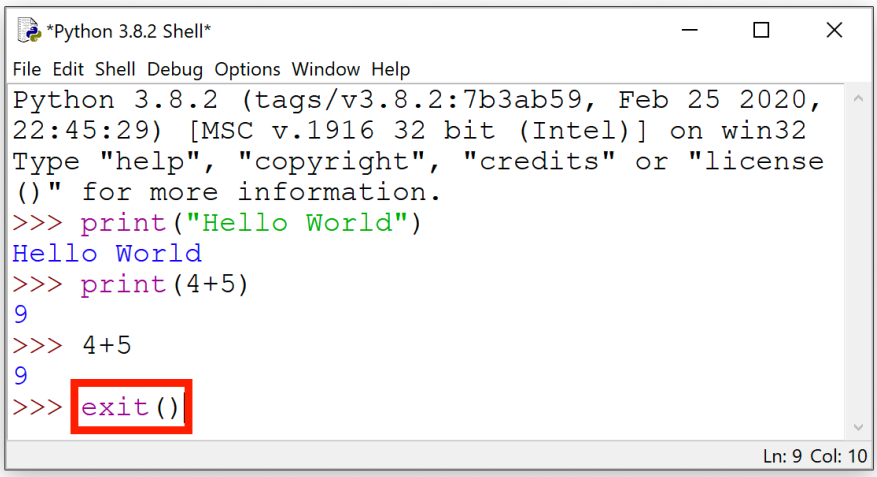
STEP - 4

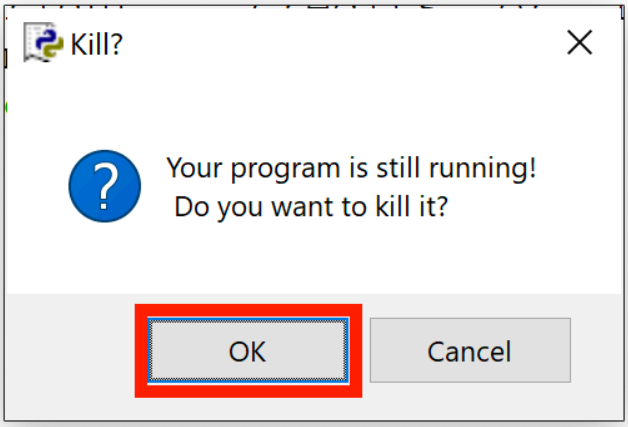
As we discussed before, that **interpreter mode** is for **testing** and **learning**.

That's why if we **skip print()** function, even after that we get our output.

STEP - 5

To get exit from IDLE window, run **exit()** function.





STEP - 6

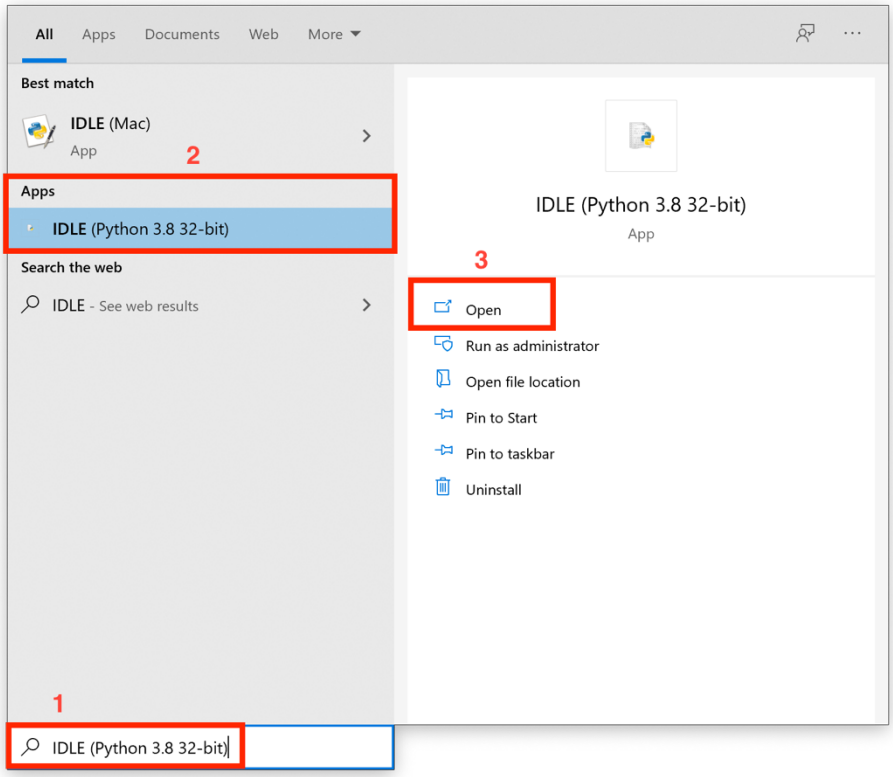
Click on **OK** button to completely close IDLE window.

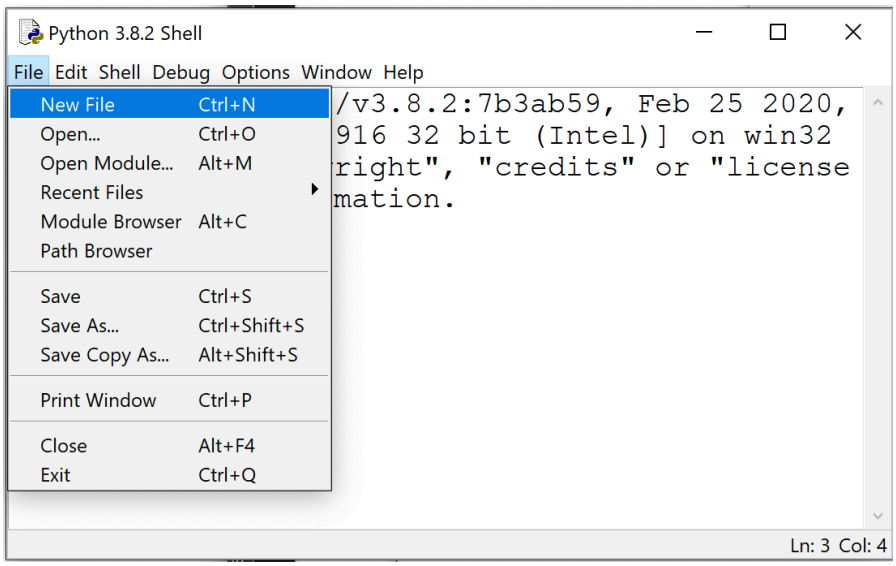
B. Using Script Mode

- **Script mode** એટલે કે પ્રોગ્રામ સંપૂર્ણ રીતે લખાયા બાદ રન થવો.
- Script mode માં આપડે કોઈ બીજી પ્રોગ્રામિંગ language ની જેમ એક file બનાવીને તેમાં coding કર્યા બાદ તે file ને રન કરવાની હોય છે. જેથી આખા પ્રોગ્રામનું આઉટપુટ એકસાથે last માં આપણને જોવા મળે છે.
- આ mode ને મૂળભૂત mode એટલે કે developing ના main mode તરીકે ઉપયોગ માં લેવામાં આવે છે.
- પાયથન પ્રોગ્રામની એક file બનાવીને તે file ને રન કરવાથી આપણને output મળે છે. તો આ પાયથન ની file ને **script** તરીકે પણ ઓળખવામાં આવે છે.
- આપડે **IDLE** ની મદદથી script mode ને સમજીશું.

STEP - 1

Search **IDLE** in search bar and click on **Open** to open Python IDLE.





STEP - 2

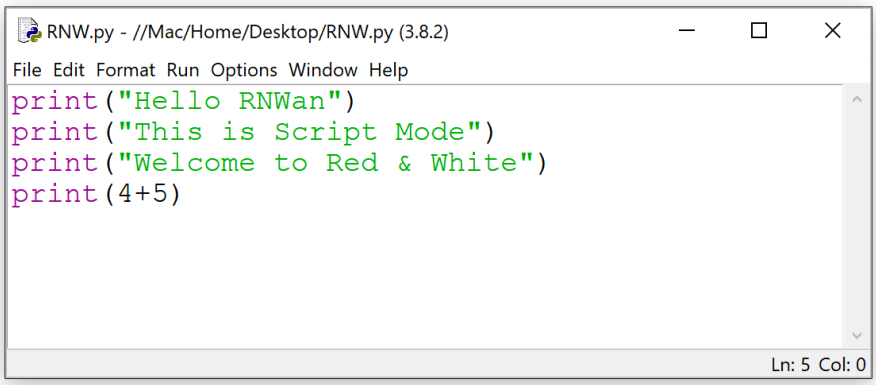
From **File** menu, select **New File** to open a new empty file in which we can code.

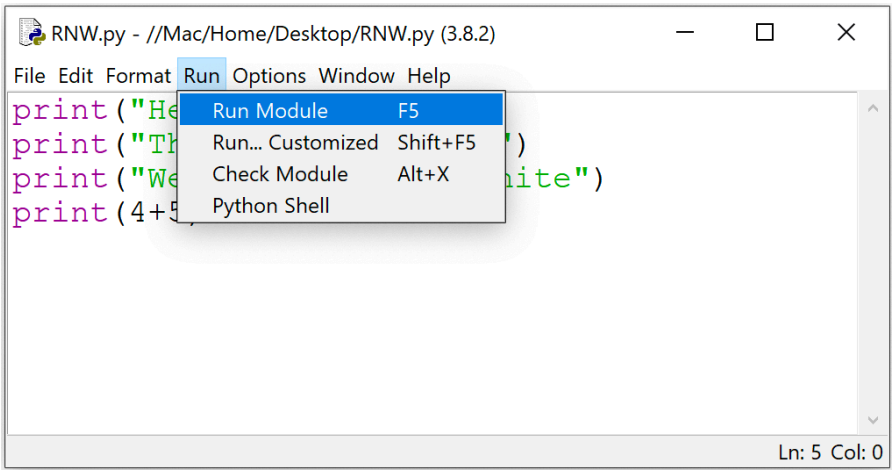
STEP - 3

Now write your all code.

Save the file with extension of **.py**

e.g. **RNW.py**





STEP - 4

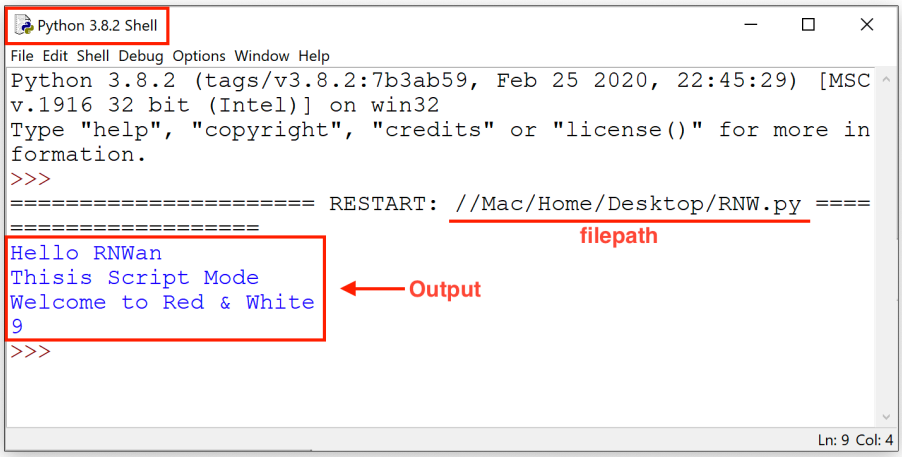
(Run Program)

From **Run** menu, select **Run Module** to run a script.
or
Press **F5** to run a script.

STEP - 5

As you can see, output always displays in a separate **Shell** window.

File path also mentioned with output.



I/O (INPUT/OUTPUT) FUNCTIONS

- પાયથન માં આઉટપુટ માટે **print()** ફંકશન નો ઉપયોગ થાય છે અને ઇનપુટ માટે **input()** ફંકશન વાપરવામાં આવે છે.
- એટલે કે પાયથન language માં I/O ફંકશનાલિટી માટે 2 functions નો ઉપયોગ કરવામાં આવે છે:
    - print()
    - input()

A. For OUTPUT

- પાયથન માં આઉટપુટ માટે **print()** ફંકશન નો ઉપયોગ થાય છે.
- print() ફંકશન ને 2 રીતે વાપરી શકાય છે.
    - For simple printing
    - For modified printing

Simple Printing:

Syntax

```
print( your_statement )
```

Example

```
print( “Hello RNWan” )
print( “Welcome” )
```

Output

```
Hello RNWan
Welcome
```

Syntax

```
print( your_statement )
```

Example

```
print( 5+3 )
```

Output

```
8
```

Modified Printing:

- By default, **print()** ફંકશન નો ઉપયોગ કરવાથી કોઈ પણ line ના આઉટપુટ માં છેલ્લે **new line (\n)** append થઇ જ જાય છે.
- print() ફંકશન માં **end** નામની એક **optional property** આવે છે. જેની મદદથી આપણે new line (\n) ના behavior ને બદલી શકીએ છીએ.
- end property ને એક **placeholder** પ્રદાન કરવાનું હોય છે જે new line (\n) ની જગ્યા એ પોતાનું સ્થાન લઇ લે છે.

Syntax

```
print( statement, end="placeholder" )
```

Example

```
print( "Hello RNWan", end="&" )
print( "Welcome" )
```

Output

```
Hello RNWan&Welcome
```

B. For INPUT

- પાયથન માં ઇનપુટ માટે **input()** ફંકશન નો ઉપયોગ થાય છે.
- input() ફંકશન ને 2 રીતે વાપરી શકાય છે.
    - Input without message
    - Input with message

Input without message:

Syntax

```
variable_name = input( )
```

Example

```
RW = input()
print( Red )
```

Output

```
Red & White          # input
Red & White
```

Input with message:

Syntax

```
variable_name = input( “your_message” )
```

Example

```
RW = input( “Enter any value: ” )
print( Red )
```

Output

```
Enter any value: Red & White
Red & White
```

DATA-TYPES IN PYTHON

- પાયથન language માં નીચે જણાવેલી DataTypes available છે:
  i. int
  ii. float
  iii. complex
  iv. str
  v. bool

VARIABLES IN PYTHON

- બીજી કોઈ પ્રોગ્રામિંગ language ની જેમ પાયથન માં **variable** ને declare કરી શકાતો નથી.
- જ્યારે કોઈ value ને પહેલી વાર assign કરીએ ત્યારે જ તે variable બની જાય છે.
- Variable માં value assign કરવા માટે **assignment operator (=)** નો ઉપયોગ થાય છે.

Example

```
R = 10
W = 10
```

- Variable માં value assign કરતી વખતે datatype પણ લખી શકાતો નથી.
- પાયથન માં variable ની datatype value પ્રમાણે automatic decide થઇ જાય છે.
- અને છતાં આપણે value assign થઇ ગયા બાદ પણ datatype બદલી શકીએ છીએ.

Example

```
R = 10          // R is of type int
R = "GIM"       // R is of type str
```

- **String** variables ને **single quotes**, **double quotes** અને **triple quotes** વડે declare કરી શકીએ છીએ.

Example

```
course = 'GIM'
course = "GIM"
course = '''GIM'''
course = """GIM"""
```

OPERATORS IN DART

- Below are operators that are available in Python:

  A. Arithmetic Operators
  B. Comparison Operators
  C. Assignment Operators
  D. Logical Operators
  E. Identity Operators
  F. Membership Operators

A. ARITHMETIC OPERATORS

Operator	Meaning
+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulo
**	Exponentiation
//	Floor Division

B. COMPARISON OPERATORS

Operator	Meaning
==	Equal
!=	Not equal
>	Greater Than
<	Less Than
>=	Greater Than or Equal To
<=	Less Than or Equal To

C. ASSIGNMENT OPERATORS

Operator	Meaning
=	Assign value
+=	Increment and assign
-=	Subtract and assign
*=	Multiply and assign
/=	Divide and assign
%=	Modulo and assign
//=	Floor division and assign
**=	Exponentiation and assign

D. LOGICAL OPERATORS

Operator	Meaning
and	True if both statements are true
or	True if one of the statements is true
not	Reverse the result, returns False if the result is true

E. IDENTITY OPERATORS

Operator	Meaning
is	Returns True if both variables are the same object
is not	Returns True if both variables are not the same object

F. MEMBERSHIP OPERATORS

Operator	Meaning
in	Returns True if a sequence with the specified value is present in the object Operator
not in	Returns True if a sequence with the specified value is not present in the object

TYPE CASTING CONSTRUCTORS

- ઘણીવાર એવી પરિસ્થિતિ આવે છે જ્યારે આપણે કોઈ variable ને નીકશ્ચિત datatype આપવો પડતો હોય છે. તો એ આપણે **casting constructors** ની મદદથી કરી શકીએ છીએ.

NOTE:
- **EVERYTHING IS OBJECT IN PYTHON.**

- પાયથન object-orientated language છે, અને જેથી કરીને datatype ને આપણે class દ્વારા define કરી શકીએ છીએ.
- તેથી **constructor functions** ની મદદથી casting કરી શકીએ છીએ:
    - int()
    - float()
    - str()
    - complex()
    - bool()

A. int()

Example

```
x = int(1)        # x will be 1
y = int(2.8)      # y will be 2
z = int("3")      # z will be 3
```

B. float()

Example

```
x = float(1)      # x will be 1.0
y = float(2.8)    # y will be 2.8
z = float("3")    # z will be 3.0
w = float("4.2")  # w will be 4.2
```

C. str()

Example

```
x = str("s1")          # x will be 's1'
y = str(2)             # y will be '2'
z = str(3.0)           # z will be '3.0'
```

D. complex()

Example

```
x = complex("6")       # x will be (6+0j)
y = complex(7j)        # y will be (7j)
z = complex(5.2)       # z will be (5.2+0j)
```

E. bool()

Example

```
a = bool(15)           # a will be True
b = bool("Hello")      # b will be True
c = bool(3.2)          # c will be True
d = bool(0)            # d will be False
e = bool(False)        # e will be False
f = bool(None)         # f will be False
g = bool("")           # g will be False
```

id() & type() FUNCTIONS

A. type() function

- કોઈ પણ object નો **datatype** મેળવવા માટે **type()** ફંકશનનો ઉપયોગ કરવામાં આવે છે.

Example

```
RED = 100
print( type(RED) )

WHITE = 15.2
print( type(WHITE) )

GIM = 'Best Course'
print( type(GIM) )

RNW = True
print( type(RNW) )

WEB = 7j
print( type(WEB) )
```

Output

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
<class 'complex'>
```

B. id() function

- **id()** ફંકશન એ કોઈ પણ object ની એક **unique id** બતાવે છે.
- પાયથન માં બધા જ objects ની પોતાની unique id હોય છે.
- જ્યારે કોઈ object બને છે ત્યારે જ તેને id assign થઇ જતી હોય છે.
- id એ object નું **memory address** છે, જેની value જ્યારે પણ program ને રન કરવામાં આવે ત્યારે અલગ-અલગ આવે છે.

Example

```
RED = 100
WHITE = 15.2
GIM = 'Best Course'
RNW = True
WEB = 7j

print( id(RED) )
print( id(WHITE) )
print( id(GIM) )
print( id(RNW) )
print( id(WEB) )
```

Output

```
4305393408
4308755984
4311279472
4305170344
4311227568
```

#2 DATATYPE IN DETAIL

STRING DATATYPE

- **String** variables ને **single quotes**, **double quotes** અને **triple quotes** વડે declare કરી શકીએ છીએ.
- જેમાં single quotes અને double quotes નો અર્થ એકસરખો જ થાય છે. એટલે કે **'Hello'** અને **"Hello"** બંને એક સમાન જ ગણાય છે.

Example

```
x = 'Hello'
y = "Hello"
```

- જ્યારે **triple quotes** ની મદદથી **Multiline string** બનાવી શકાય છે. Triple quotes ને single અથવા double quotes ના symbol થી ત્રણ વાર લખીને બનાવી શકાય છે.

Example – Using single quotes

```
RNW = '''Welcome to Red & White Group of Institutes.
This is Multimedia Education department.
You are learning Python.'''
```

Example – Using double quotes

```
RNW = """Welcome to Red & White Group of Institutes.
This is Multimedia Education department.
You are learning Python."""
```

STRING FORMATTING

- પાયથન language માં string અને numbers ને નીચે જણાવેલ example ની જેમ combine કરી શકતા નથી.

Example

```
age = 30
txt = "My name is Python, I am " + age
print(txt)
```

- ઉપરનું example આપણને **error** આપે છે.
- પણ આપણે string અને numbers ને અલગ-અલગ 3 methods ની મદદથી combine કરી શકી છીએ:
    - Using **format()** method
    - Using **format specifier placeholders**
    - Using **f-string**

A. Using format() method

- **format()** method ને **arguments** આપવા પડે છે કે જેને આપણે string ની અંદર મુકવા છે. અને આ આપેલા arguments એ string માં દર્શાવવામાં આવેલા **placeholder { }** ની જગ્યા પર મુકાય જાય છે.

Example

```
age = 30
txt = "My name is Python, I am { }"
print(txt.format(age))
```

Output

```
My name is Python, I am 30
```

- format() method ને **ગમે તેટલા arguments** આપી શકીએ છીએ અને તે **ક્રમ-wise** પોતાના placeholders ની જગ્યા પર મુકાય જાય છે.

Example

```
quantity = 100
branch = 5
RNW = "Red & White Group of Institutes have total of { } branches
and more than { } courses available."
print(RNW.format(branch, quantity))
```

Output

Red & White Group of Institutes have total of **5** branches and more than **100** courses available.

- આપણે **index numbers** નો ઉપયોગ પણ કરી શકીએ છીએ જેથી બધા arguments પોતાના નિચ્છીત placeholder સ્થાન પર મુકાય જાય.

Example

```
quantity = 100
branch = 5
RNW = "Red & White Group of Institutes have total of { 1 }
branches and more than { 0 } courses available."
print(RNW.format(quantity, branch))
```

Output

Red & White Group of Institutes have total of **5** branches and more than **100** courses available.

B. Using format specifier placeholders

- પાયથન language માં C language ની format specifier વાળી syntax પ્રમાણે "%" operator ના ઉપયોગ થી string formatting કરી શકીએ છીએ.
- E.g. **%d** એ integer ને દર્શાવવા માટે અને **%s** string ને દર્શાવવા માટે વપરાય છે.

Example

```
name = "RNWan"
print("Hello, %s" % name)
```

Output

```
Hello, RNWan
```

- બે અથવા વધારે arguments માટે નીચે બતાવેલા example પ્રમાણે parantheses નો ઉપયોગ કરવો.

Example

```
name = "RNW"
age = 12
print("%s is %d years old." % (name, age))
```

Output

```
RNW is 12 years old.
```

- પાયથન બધા જ objects ને **%s** વડે પણ દર્શાવી શકાય છે, પેસી ભલે તે object કોઈ પણ datatype ધરાવતો હોય.

Example

```
name = "RNW"
courses = [ "GIM", "WEB", "FLUTTER", "PYTHON" ]    // list datatype
print("%s has %s courses." % (name, courses))
```

Output

```
RNW has [ "GIM", "WEB", "FLUTER", "PYTHON" ] courses.
```

C. Using f-string

- પાયથન language માં string formatting ને **f-string** ની મદદથી વધારે સરળ બનાવી શકાય છે.
- **f-string** પાયથન **3.6** version માં introduce કરવામાં આવેલી હતી.
- f-string માં પણ format() method ની જેમ **placeholders { }** નો ઉપયોગ કરવામાં આવે છે. પણ આ placeholders { } ની અંદર ફરજિયાતપણે જેતે object અથવા variable ને દર્શાવવા જરૂરી છે.
- f-string ને નીચે આપેલા example પ્રમાણે lower **f** અથવા તો upper **F** વડે દર્શાવી શકાય છે.

Example

```
name = "RNW"
age = 12
print(f"{name} is {age} years old.")
print(F"{name} is {age} years old.")
```

Output

```
RNW is 12 years old.
RNW is 12 years old.
```

STRING MANIPULATION

- પાયથન language માં **character** datatype available નથી. પણ પાયથનમાં એક single character એક **string** જ ગણવામાં આવે છે જેની **length 1** હોય છે.
- String ને પાયથન ની અંદર બાકી ઘણીબધી languages ની જેમ **characters ના array** ની જેમ જ treat કરવામાં આવે છે.
- તેથી string ના elements ને access કરવા **માટે square brackets [ ]** નો ઉપયોગ થાય છે.

Syntax

```
string[ index ]
```

Example

```
RNW = "Welcome to Red & White"
print( RNW[1] )
print( RNW[-2] )        # negative index starts from(-1) last position
```

Output

```
e
t
```

- String માંથી કોઈ **specific characters** જોઈતા હોય તો આપણે string **slicing** નો ઉપયોગ કરી શકીએ છીએ.
- Slicing કરતી વખતે **starting index** અને **ending index** ને **square brackets [ ]** માં comma, થી separate કરીને specify કરવું પડે છે.

Syntax

```
string[ starting_index : ending_index ]
```

Example

```
RNW = "Welcome to Red & White"
print( RNW[11 : 14] )
print( RNW[-5 : -1] )
```

Output

```
Red
Whit
```

- Slicing કરતી વખતે starting અને ending index ની સાથે સાથે **stepping value** પણ provide કરી શકીએ છીએ. જેટલી stepping value હશે તેટલા character, slicing કરતી વખતે **skip** થતા રહેશે.

Example

```
RNW = "Welcome to Red & White"
print( RNW[0 : 7 : 2] )
```

Output

```
Wloe
```

TYPES OF COMMENTS

- પાયથનમાં 2 પ્રકારની comments લખી શકીએ છીએ:
  - Single-line Comment
  - Multi-line Comment

A. Using Single-line Comment

- Single-line Comment કરવા માટે **#** નો ઉપયોગ થાય છે.

Example

```
R = 10
# R = R - 3
print( R )
```

Output

```
10
```

B. Using Multi-line Comment

- Multi-line Comment કરવા માટે **triple quotes** (‘‘‘) નો ઉપયોગ થાય છે.

Example

```
'''Red & White Group of Institutes,
Located in Surat,
Gujarat, India
'''
```

COLLECTION DATATYPES

- પાયથન language માં નીચે જણાવેલી Collection DataTypes available છે:
  - **A.** List
  - **B.** Tuple
  - **C.** Set
  - **D.** Dictionary

A. List

- **List** એ values નો **સમૂહ** છે જે ordered ક્રમમાં હોય છે અને જેને બદલી પણ શકાય છે.
- List ની અંદર duplicate members પણ હોય શકે છે.
- જેમ બાકીની કોઈ પ્રોગ્રામિંગ languages માં **Array** આવે છે તે જ રીતે પાયથન માં **List** આવે છે.
- Array એટલે કે એક સરખી datatype ધરાવતી values નો સમૂહ. જ્યારે List ની અંદર એકસાથે કોઈ પણ datatype ધરાવતી values ને સંગ્રહિત કરી શકીએ છીએ.
- List ને **square brackets [ ]** સાથે લખવામાં આવે છે.

Syntax

```
variable = [ items ]
```

Example

```
courses = [ "GIM", "WEB", "Android", "Flutter", "Python" ]
print( courses )
```

Output

```
[ 'GIM', 'WEB', 'Android', 'Flutter', 'Python' ]
```

- List ના બધા જ members અથવા items ને index ની મદદથી access કરી શકાય છે.

Example

```
courses = [ "GIM", "WEB", "Android", "Flutter", "Python" ]
print( courses[2] )
```

Output

```
Android
```

- List ના બધા જ members અથવા items ને index ની મદદથી access કરી શકાય છે.
- List સાથે **slicing** operation પણ કરી શકાય છે જેમ string સાથે slicing perform થતું હતું તે રીતે.

Example

```
courses = [ "GIM", "WEB", "Android", "Flutter", "Python" ]
print( courses[2 : 4] )
print( courses[ : 4] )
print( courses[-4 : -2] )
print( courses[ : : 2] )
```

Output

```
[ 'Android', 'Flutter' ]
[ 'GIM', 'WEB', 'Android', 'Flutter' ]
[ 'WEB', 'Android' ]
[ 'GIM', 'Android', 'Python' ]
```

- List એ **mutable** datatype છે એટલે કે તેમાં **changes થઇ શકે છે** જેમ કે કોઈ item ને change કરવી, delete કરવી તથા નવી item insert કરવી.
- કોઈ પણ specific item ની value change કરવા માટે index નો ઉપયોગ થાય છે.

Example

```
courses = [ "GIM", "WEB", "Android", "Flutter", "Python" ]
courses[1] = "IOS"
print( courses )
```

Output

```
[ 'GIM', 'IOS', 'Android', 'Flutter', 'Python' ]
```

- List પણ પાયથનમાં એક object જ છે. તેથી List ઘણીબધી methods ધરાવે છે જે ખૂબ જ ઉપયોગી બની રહે છે:

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Example

```
courses = [ "GIM", "WEB", "Android", "Flutter" ]
courses.append("Python")
print( courses )
```

Output

```
[ 'GIM', 'IOS', 'Android', 'Flutter', 'Python' ]
```

B. Tuple

- Tuple એ values નો સમૂહ છે જે ordered ક્રમમાં હોય છે પણ તેને બદલી શકાતી નથી.
- Tuple ની અંદર duplicate members પણ હોય શકે છે.
- Tuple ને **round brackets ( )** સાથે લખવામાં આવે છે.

Syntax

```
variable = ( items )
```

Example

```
courses = ( "GIM", "WEB", "Android", "Flutter", "Python" )
print( courses )
```

Output

```
( 'GIM', 'WEB', 'Android', 'Flutter', 'Python' )
```

- Tuple ના બધા જ members અથવા items ને index ની મદદથી access કરી શકાય છે.
- Tuple એ **immutable** datatype છે એટલે કે તેમાં **changes થઇ શકતો નથી** જેમ કે કોઈ item ને change કરવી, delete કરવી તથા નવી item insert કરવી તે શક્ય નથી.

Example

```
courses = ( "GIM", "WEB", "Android", "Flutter", "Python" )
courses[1] = "IOS"                # This will raise an error
print( courses )
```

Output

```
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    courses[1] = "IOS"
TypeError: 'tuple' object does not support item assignment
```

- જો એક એવી **tuple** બનાવવી છે કે જે **એક જ item** ધરાવતી હોય તો round brackets માં તે item ની દર્શાવ્યા બાદ એક **comma (,)** ઉમેરવો જરૂરી છે, નહીંતર પાયથન તેને એક tuple તરીકે treat નહીં કરી શકે.

Example

```
RNW = ("GIM", )
print( type(RNW) )

RNW = ("GIM" )                    # Not a tuple
print( type(RNW) )
```

Output

```
<class 'tuple'>
<class 'str'>
```

- Tuple પણ પાયથનમાં એક object જ છે. Tuple ફક્ત બે જ methods ધરાવે છે જે ખૂબ જ ઉપયોગી બની રહે છે:

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

C. Set

- **Set** એ values નો સમૂહ છે જે **unordered** ક્રમમાં હોય છે અને set indexing ધરાવતો નથી.
- Set ની અંદર **duplicate members** આવી શકતા નથી.
- Set ને **curly brackets { }** સાથે લખવામાં આવે છે.

Syntax

```
variable = { items }
```

Example

```
courses = ( "GIM", "WEB", "GIM", "WEB", "Python" )
print( courses )
```

Output

```
{ 'GIM', 'WEB', 'Python' }
```

- Set પણ પાયથનમાં એક object જ છે. તેથી Set ઘણીબધી methods ધરાવે છે જે ખૂબ જ ઉપયોગી બની રહે છે:

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	inserts the symmetric differences from this set and another
update()	Update the set with the union of this set and others

- જ્યારે set બની જાય છે, પછી તેની items બદલી શકાતી નથી પણ નવી items add કરી શકાય છે.

Example

```
courses = ( "GIM", "WEB", "Python" )
courses.add("Flutter")
print( courses )
```

Output

```
{ 'GIM', 'WEB', 'Flutter', 'Python' }
```

D. Dictionary

- Dictionary એ values નો સમૂહ છે જે unordered ક્રમમાં હોય છે અને changable પણ હોય છે. Dictionary indexing ધરાવે છે અને તેમાં **key -value pairs** હોય છે.
- Dictionary ની અંદર **duplicate keys** આવી શક્તી નથી.
- Dictionary ને **curly brackets { }** સાથે લખવામાં આવે છે.

Syntax

```
variable = { key1 : value1, key2 : value2, ... , keyN : valueN }
```

Example

```
RNW = {
        "Name" : "Red & White Group of Institutes",
        "Year" : 2008,
        "Branches" : 5
}
print( RNW )
```

Output

```
{ 'Name': 'Red & White Group of Institutes', 'Year': 2008, 'Branches': 5 }
```

- Dictionary ની કોઈ પણ value ની access કરવા માટે **key** નો ઉપયોગ થાય છે:

Example

```
print( RNW["Name"] )
```

Output

```
Red & White Group of Institutes
```

- કોઈ નવી value ની dictionary માં insert કરવા માટે નવી **key** ની મદદ લેવી પડે છે:

Example

```
RNW["Courses"] = 120
print(RNW)
```

Output

```
{ 'Name': 'Red & White Group of Institutes', 'Year': 2008, 'Branches': 5,
'Courses': 120 }
```

- Dictionary માં આપણે **nested dictionary** પણ બનાવી શકીયે છીએ:

Example

```
Courses = {
        "GIM" : {
                "Name" : "Graduate in Multimedia",
                "Duration" : "12 Months",
                "Pre-requirement" : "Not any"
        },
        "Flutter" : {
                "Name" : "The Mobile App Development with Flutter",
                "Duration" : "8 Months",
                "Pre-requirement" : "C & C++"
        },
        "Python" : {
                "Name" : "The Python Course",
                "Duration" : "6 Months",
                "Pre-requirement" : "C & C++"
        }
}
print( Courses )
```

Output

```
{ 'GIM' : { 'Name' : 'Graduate in Multimedia', 'Duration' : '12 Months',
'Pre-requirement' : 'Not any' }, 'Flutter' : { 'Name' : 'The Mobile App
Development with Flutter', 'Duration' : '8 Months', 'Pre-requirement' : 'C
& C++' }, 'Python' : { 'Name' : 'The Python Course', 'Duration' : '6 Months',
'Pre-requirement' : 'C & C++' } }
```

- Dictionary પણ પાયથનમાં એક object જ છે. તેથી Dictionary ઘણીબધી methods ધરાવે છે જે ખૂબ જ ઉપયોગી બની રહે છે:

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitems()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

MUTABILITY OF LIST & TUPLE

- List અને Tuple માં મોટો તફાવત એ છે કે **List** એ **mutable** છે જયારે **Tuple** એ **immutable** છે.
- એટલે કે List માં changes થઇ શકે છે જેમ કે કોઈ item ને change કરવી, delete કરવી તથા નવી item insert કરવી. જયારે Tuple માં આ બધું કરી શકાતું નથી.

Example - List Manipulation

```
courses = [ "GIM", "WEB", "Android", "Flutter", "Python" ]
courses[1] = "IOS"
print( courses )
```

Output

```
[ 'GIM', 'IOS', 'Android', 'Flutter', 'Python' ]
```

Example - Tuple Manipulation

```
courses = ( “GIM”, “WEB”, “Android”, “Flutter”, “Python” )
courses[1] = “IOS”                # This will raise an error
print( courses )
```

Output

```
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    courses[1] = "IOS"
TypeError: 'tuple' object does not support item assignment
```

SHALLOW & DEEP COPY

- પાયથન માં **assignment operator (=)** એ કોઈ પણ object ની **copy** બનાવે છે. આપણે વિચારીશું કે એક નવો object બની જાય છે, પણ તેવું થતું નથી.
- Assignment operator ફક્ત એક નવો **variable** બનાવે છે જે original object નો **reference share** કરે છે.

Example

```
courses = [ ‘CCC’, ‘Tally’, ‘Advance Excel’ ]
our_courses = courses
courses.pop()                    # pop() removes last element
print(courses)
print(our_courses)
```

Output

```
[ ‘CCC’, ‘Tally’ ]
[ ‘CCC’, ‘Tally’ ]
```

- ઉપરના example પરથી સાબીત થાય છે કે **Assignment operator** ફક્ત object નો **reference share** કરે છે.
- એટલે જો હવે કોઈ પણ પ્રકારનું operation **courses** કે **our_courses** પર કરવામાં આવશે તો બંને માં તેની અસર જોવા મળશે. તો આવા પ્રકારની copy ને **shallow copy** કહેવામાં આવે છે.
- **Deep copy** એક નવો object બનાવે છે અને તેમાં original object માં રહેલા તમામ **elements ની કોપી** ને **add** કરે છે.
- List અથવા તો Tuple ની **deep copy** કરવા માટે તેમની built-in method **copy()** નો ઉપયોગ કરવામાં આવે છે.

Example

```
courses = [ 'CCC', 'Tally', 'Advance Excel' ]
our_courses = courses.copy()
courses.pop()                    # pop() removes last element
print(courses)
print(our_courses)
```

Output

```
[ 'CCC', 'Tally' ]
[ 'CCC', 'Tally', 'Advance Excel' ]
```

- ઉપરના example પરથી આપણે જોઈ શકીયે છીએ કે **copy()** method ની મદદથી નવો object બને છે જે original object ના બધા જ members ની copy અલગ થી બનાવે છે.

TYPE CASTING CONSTRUCTORS FOR COLLECTION DATATYPE

- Collection datatypes ને પણ **type casting** કરી શકાય છે. Type casting કરવા માટે નીચે જણાવેલા **type casting constructors** ને ઉપયોગ માં લઇ શકીએ છીએ:
    - list()
    - tuple()
    - set()
    - frozenset()
    - dict()

A. list()

- કોઈ પણ **object** ને List માં **convert** કરવા માટે **list()** type casting convertor નો ઉપયોગ થાય છે.

Example

```
my_tuple = ( 12, 34, 56, 23, 56 )
my_list = list(my_tuple)
print(my_list)
```

Output

```
[ 12, 34, 56, 23, 56 ]
```

B. tuple()

- કોઈ પણ **object** ને Tuple માં **convert** કરવા માટે **tuple()** type casting convertor નો ઉપયોગ થાય છે.

Example

```
my_list = [ 12, 34, 56, 23, 56 ]
my_tuple = tuple(my_list)
print(my_tuple)
```

Output

```
( 12, 34, 56, 23, 56 )
```

C. set()

- કોઈ પણ **object** ને Set માં **convert** કરવા માટે **set()** type casting convertor નો ઉપયોગ થાય છે.

Example

```
my_list = [ 12, 34, 56, 23, 56, 12, 90 ]
my_set = set(my_list)
print(my_set)
```

Output

```
{ 12, 34, 56, 23, 90 }
```

D. frozenset()

- frozenset() આપણને **immutable set** આપે છે. એટલે કે એક એવો set જેમાં આપણે કઈ પણ change કરી શક્તા નથી. અને frozenset એ **unordered** ક્રમમાં હોય છે.
- frozenset સાથે simpe set પર કરી શકતા બધા જ operations perform કરી શકીએ છીએ.
- કોઈ પણ **object** ને frozenset માં **convert** કરવા માટે **frozenset()** type casting convertor નો ઉપયોગ થાય છે.

Example

```
my_tuples = ( ('Name', 'RNW'), ('Year', 2008), ('Branches', 5) )
my_dict = dict(my_tuples)
print(my_dict)
```

Output

```
{ 'Name': 'RNW', 'Year': 2008, 'Branches': 5 }
```

E. dict()

- કોઈ પણ **object** ને Dictionary માં **convert** કરવા માટે **dict()** type casting convertor નો ઉપયોગ થાય છે.

Example

```
my_tuples = ( ('Name', 'RNW'), ('Year', 2008), ('Branches', 5) )
my_dict = dict(my_tuples)
print(my_dict)
```

Output

```
{ 'Name': 'RNW', 'Year': 2008, 'Branches': 5 }
```

del KEYWORD

- **del** keyword નો ઉપયોગ object ને **delete** કરવા માટે થાય છે. પાયથન માં બધુંજ object છે, તેથી del keyword ને કોઈ પણ object આથવા variable ને delete કરવા માટે ઉપયોગ માં લઇ શકાય છે.

Example

```
RNW = "GIM"
del RNW
print(RNW)
```

Output

```
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    Print(RNW)
NameError: name 'RNW' is not defined
```

Example

```
courses = [ "GIM", "WEB", "Android", "Flutter", "Python" ]
del courses[1]
print(courses)
```

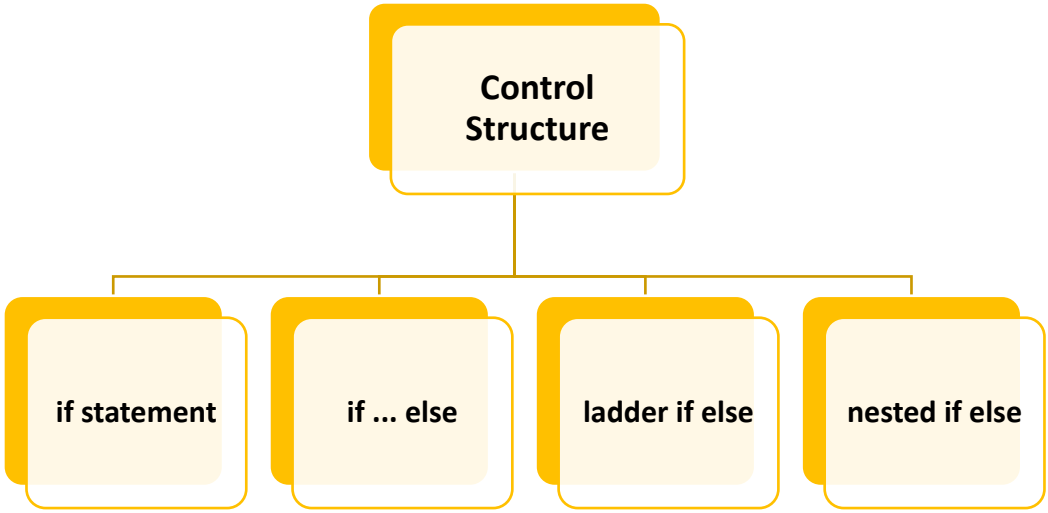
Output

```
[ 'GIM', 'Android', 'Flutter', 'Python' ]
```

- del keyword ની મદદથી list, tuple, set, dictionary જેવા collection datatypes ને પણ delete કરી શકીએ છીએ.

#3 CONTROL STRUCTURE & LOOPING

TYPES OF CONTROL STRUCTURE



- Below are types of control structure that are available in Python:
  - **A.** if Statement
  - **B.** if … else
  - **C.** ladder if … else
  - **D.** nested if … else

- પાયથનમાં કોઈ પણ block નો **scope** બતાવવા માટે curly brackets **{ }** ની જગ્યાએ **indentation (whitespace)** નો ઉપયોગ થાય છે.
- પાયથનમાં indentation ખુબ જ મહત્વનો ભાગ ભજવે છે. ખાસ કરીને જયારે કોઈ control structure, function અને class નો scope define કરતી વખતે જો indentation દર્શાવવાનું બાકી રહી જાય તો તે જગ્યા પર error આવતી હોય છે.
- **Indentation** સામાન્ય રીતે અમુક **whitespace** ની મદદથી બનાવી શકાય છે. પાયથનમાં સામાન્ય રીતે **2** અથવા **4** whitespace વાળું Indentation વધારે ઉપયોગમાં લેવામાં આવે છે.

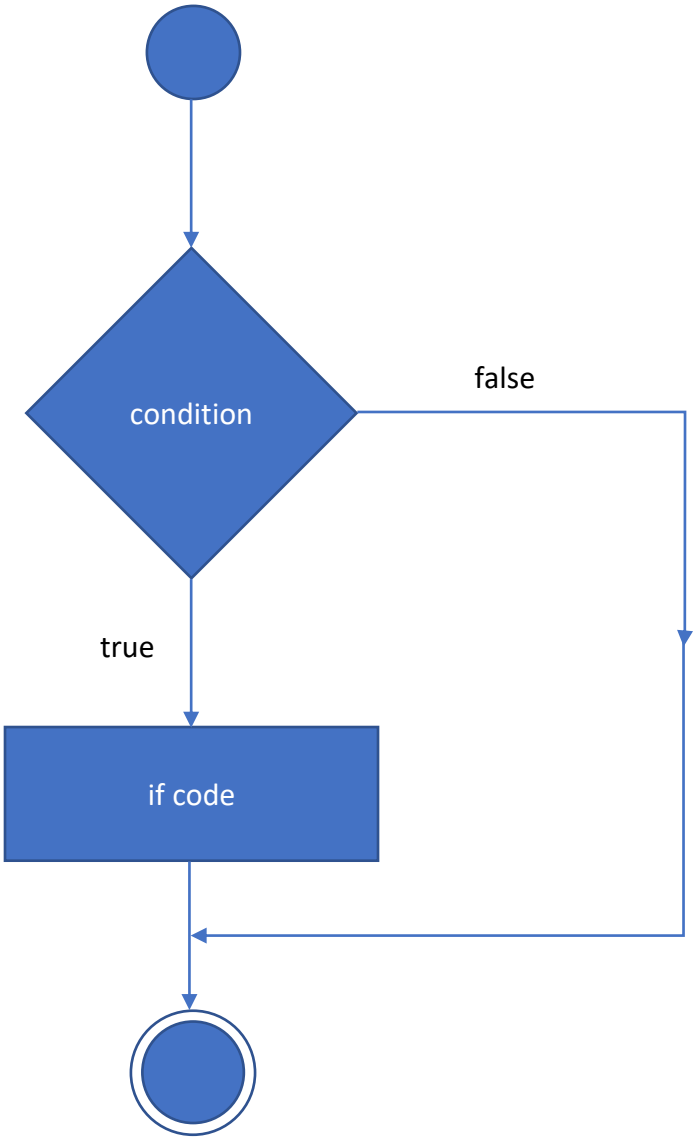
NOTE:
- આખા **Program** અથવા **Script** ની અંદર હરેક જગ્યા પર એકસરખું જ **Indentation** વાપરવું.

A. if Statement

Syntax

```
if condition:
----# Statements
```

Control Flow



Example

```
if 5>3:
    print("True Statement.")
```

Output

```
True Statement.
```

B. if … else

Syntax

```
if condition:
----# True Part Statements
else
----# False Part Statements
```

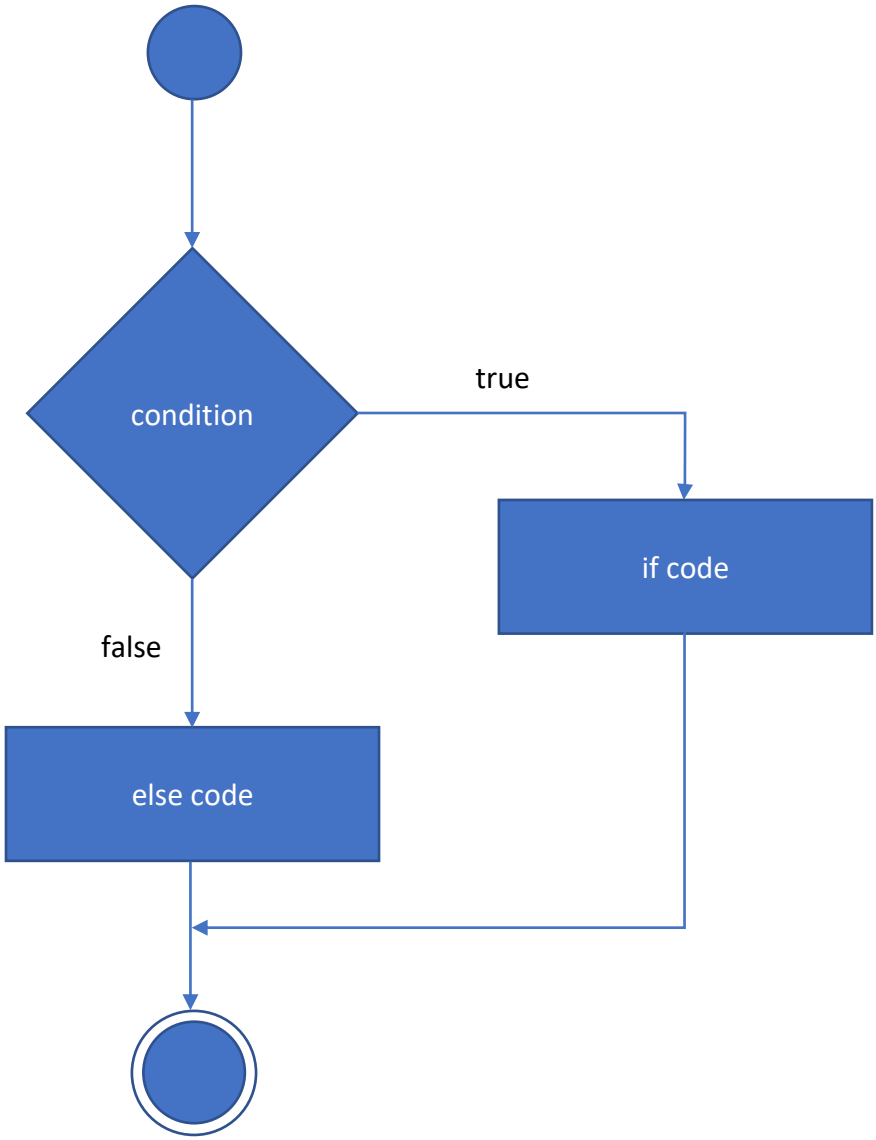
Example

```
if 5<3:
    print("True Statement.")
else:
    print("False Statement.")
```

Output

```
False Statement.
```

Control Flow



C. ladder if else

Syntax

```
if condition1:
----# Statements
elif  condition2:
----# Statements
elif condition3:
----# Statements
else                    # here, else block is optional.
----# Statements
```

Example

```
age = 23
if age>18:
    print("You can vote.")
elif age<18:
     print("You cannot vote.")
elif age==18:
    print("You can also vote.")
```

Output

```
You can vote.
```

D. nested if else

Syntax

```
if condition1:
----if condition2:
--------# Statements
else
----if condition3:
--------# Statements
```

Example

```
a=5
b=3
c=6
if a>b:
    if a>c:
        print("a is max")
    else
        print("c is max")
else
    if b>c:
        print("b is max")
    else
        print("c is max")
```

Output

```
c is max
```

- જો કોઇ **એક line** નું statement હોય તો આપણે તેને એક જ line માં if statement ની સાથે જ દર્શાવી શકીયે છીએ.
- આ રીતે એક જ લઇને માં દર્શાવવામાં આવેલ method ને **Short hand syntax (ternary)** પણ કહેવામાં આવે છે.

Example

```
if 5>3: print("True Statement.")
```

Output

```
True Statement.
```

Example

```
R=5
W=7
print("R") if R>W else print("W")
```

Output

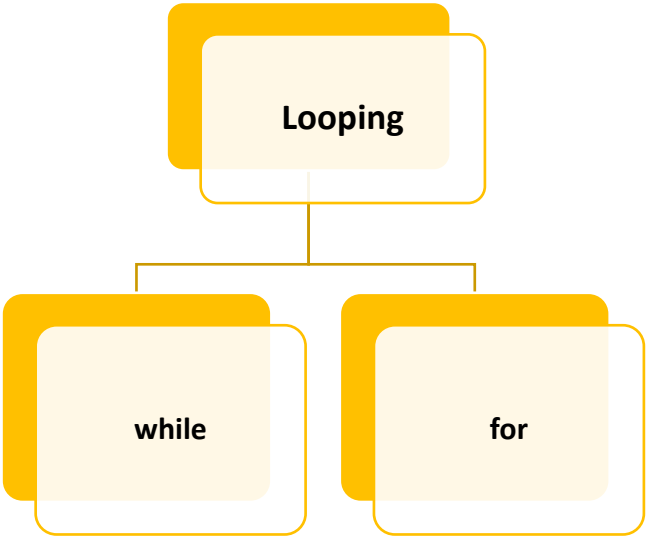
```
7
```

NOTE:

- પાયથનમાં **switch case** ઉપલબ્ધ નથી. પણ **ladder if else** ની મદદથી તેના જેવું **working** મેળવી શકાય છે.

LOOPING

- Below are types of loops that are available in Python:
  - **A.** while loop
  - **B.** for loop

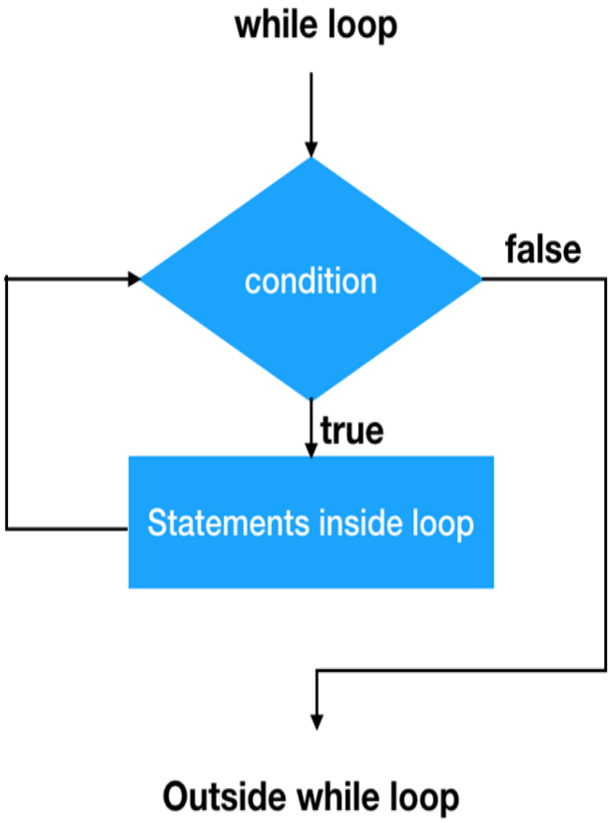


A. while loop

Syntax

```
# initialization
while condition:
----# statements
----# increment/decrement
```

Control Flow



Example

```
i = 1
while i<=10:
        print(i)
        i += 1
```

Output

```
1
2
3
4
5
6
7
8
9
10
```

NOTE:

- પાયથનમાં **increment operator (++)** અને **decrement operator (--)** ઉપલબ્ધ નથી.

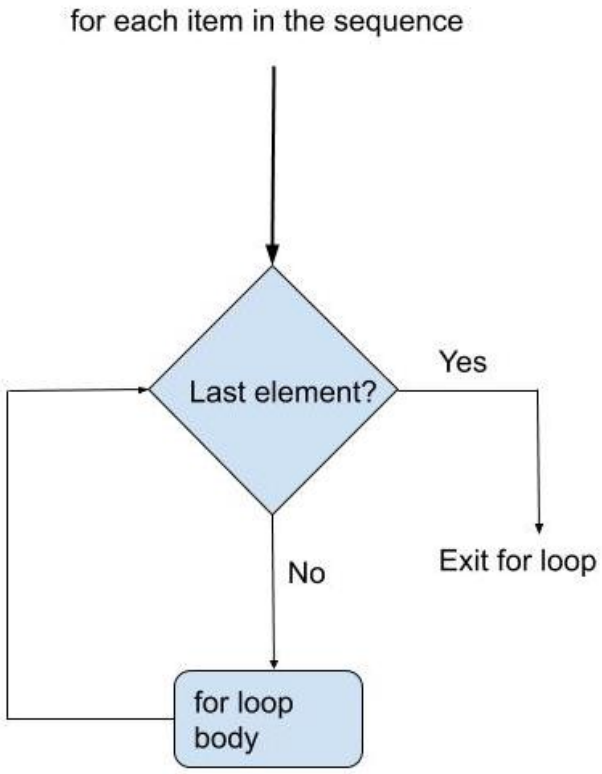
B. for loop

- for loop એ કોઈ sequence (જેમ કે list, tuple, set, dictionary અથવા string) માં iterate કરવા માટે વપરાય છે.

Syntax

```
for variable in sequence:
----Statements
```

Control Flow



Example

```
for i in [1,2,3,4,5,6,7,8,9,10]:
    print(i, end=" ")
```

Output

```
1 2 3 4 5 6 7 8 9 10
```

- for loop ની સાથે આપણે **else** block નો ઉપયોગ પણ કરી શકીયે છીએ. else block માં લખેલું coding ત્યારે જ રન થાય છે જયારે for loop execute થઇ જાય છે અને તેનું કામ પૂરું થઇ જાય છે.

Example

```
for i in [1,2,3,4,5,6,7,8,9,10]:
    print(i, end=" ")
else:
    print("\nFinally finished...")
```

Output

```
1 2 3 4 5 6 7 8 9 10
Finally finished...
```

range() FUNCTION

- કોઈ ચોક્કસ number પ્રમાણે code ને loop થી ફેરવવા માટે **range()** function નો ઉપયોગ થાય છે.
- range() function **sequence of numbers** return કરે છે જે by-default **0** થી શરુ થાય છે અને કોઈ ચોક્કસ number પર પૂરું થાય છે જે by-default **1** number નો વધારો કરતુ જાય છે.
- Return કરેલી sequence નો datatype **range** હોય છે.

Example

```
print( range(10) )
```

Output

```
range(0, 10)
```

- સામાન્ય રીતે range() function ને loop ની સાથે વધારે વાપરવામાં આવે છે.

Example

```
for i in range(6):
    print(i)
```

Output

```
0
1
2
3
4
5
```

NOTE:

- **range(6) ની values 0 થી 6 નહીં, પણ 0 થી 5 આવશે.**

- range() function ની starting value by-default 0 છે, પણ આપણે કોઈ પણ value ને starting value તરીકે રાખી શકીએ છીએ. આવું કરવા માટે આપણે range() function ને હજુ એક parameter આપવો પડે છે. For example, range(2 ,6) આપણને 2 થી 5 values આપશે.

Example

```
for i in range(2, 6):
    print(i)
```

Output

```
2
3
4
5
```

- range() function માં by-default 1 નો વધારો થતો હોય છે, પણ આપણે ત્રીજો parameter આપીને increment value ને બદલી શકીએ છીએ.

Example

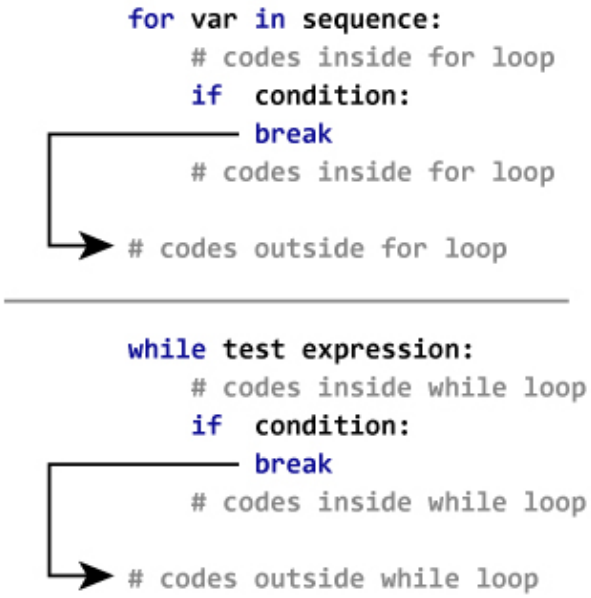
```
for i in range(2, 30, 3):
    print(i)
```

Output

```
2
5
8
11
14
17
20
23
26
29
```

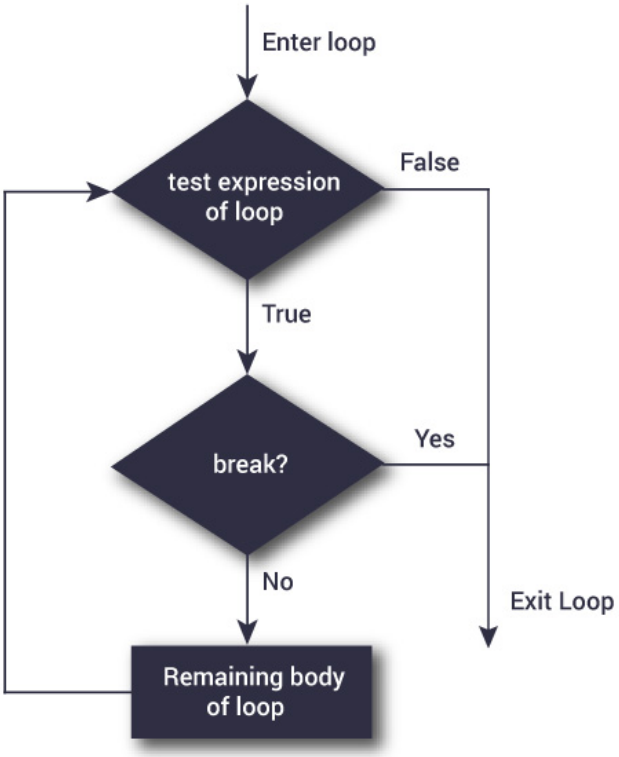
CONTROL STATEMENTS

- ઘણીવાર અમુક પરિસ્થિતિઓ એવી આવે છે કે આપડે loop નું coding skip કરવું પડે છે અથવા loop માંથી તરત જ બહાર નીકળી જવું પડતું હોય છે.
- આવી પરિસ્થિતિઓ માટે **break** અને **continue** જેવા statements નો યુઝ કરવામાં આવે છે.
- **break** statement નો યુઝ loop ને તરત જ terminate કરવા માટે થાય છે. અને program નો control loop ની બહાર આવતા statement તરફ ધકેલાય જાય છે.



A. break

Control Flow



Example

```
for i in “python”:
    if i == “h”:
        break
    print(i)
```

Output

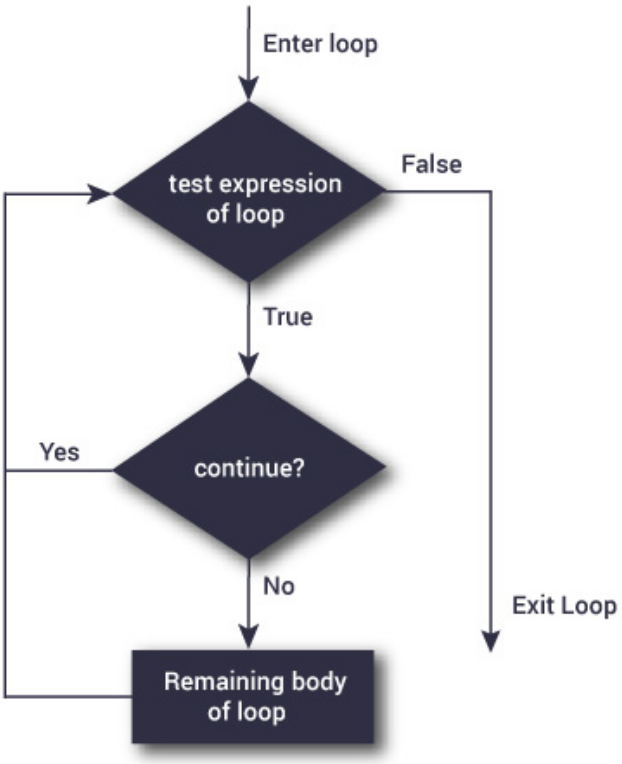
```
p
y
t
```

- **continue** statement નો ઉપ્ર હાલનું ચાલતું iteration skip કરવા માટે થાય છે.
- **continue** statement એ **for loop** સાથે જ ઉપ્ર કરી શકાય છે, જ્યારે break statement કોઈ પણ loop સાથે ઉપ્ર કરી શકાય છે.

```
for var in sequence:
    # codes inside for loop
    if  condition:
        continue
    # codes inside for loop

# codes outside for loop
```

B. continue

Control Flow



Example

```
for i in “python”:
    if i == “h”:
        continue
    print(i)
```

Output

```
p
y
t
o
n
```

C. pass

- પાયથનમાં **pass** statement એ એક **null statement** છે. Comment અને pass statement વચ્ચે તફાવત એટલો જ છે કે compiler એ comment ને ignore કરે છે જયારે pass statement ને ignore કરતુ નથી.
- છતાંપણ જયારે pass statement execute થાય છે ત્યારે કંઈ પણ થતું નથી. મતલબ કે **no operation (NOP)**.
- સામાન્ય રીતે pass ને **placeholder** ની જેમ ઉપયોગમાં લેવામાં આવે છે.
- ધારો કે આપણી પાસે એક એવી loop, function અથવા તો class છે જેને હજુ implement કરવાનું બાકી છે, પણ આપણે તેને future માં implement કરવા માંગીએ છીએ. આ પરિસ્થિતિમાં તેઓની body ને ખાલી રાખી શકાય નહીં, અને જો ખાલી રાખી તો interpreter error આપશે. તો ત્યારે આપણે pass statement નો ઉપયોગ કરી શકીએ છીએ જે એક body નો આભાસ બતાવશે કે જે કઈ પણ કરતી હશે નહીં.

Example

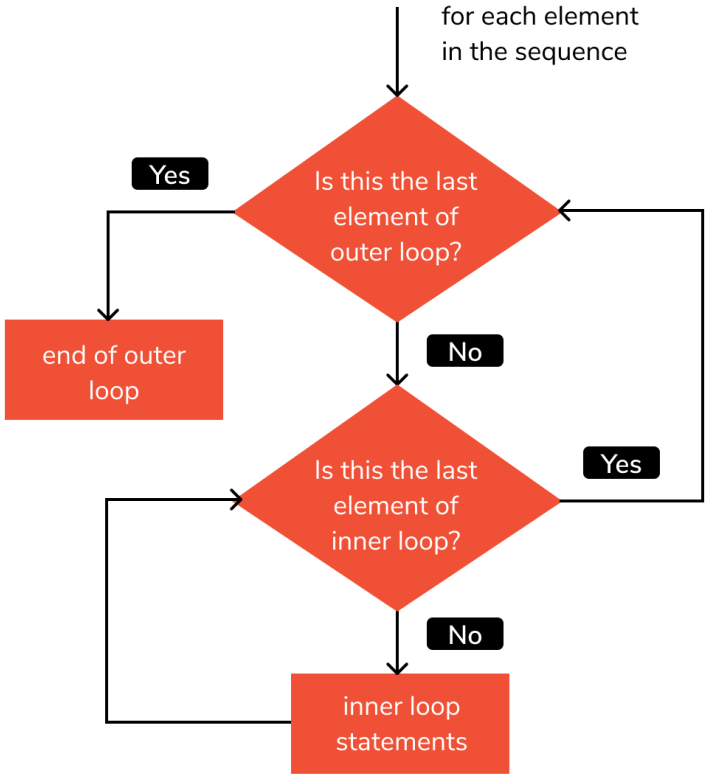
```
for i in “python”:
    pass
```

- ઉપરનાં પ્રોગ્રામ ને રન કરવાથી output માં કંઈ પણ આવતું નથી.

NESTED LOOP

- પાયથન આપણને **loop ની અંદર loop** યુઝ કરવા દે છે, જેને આપડે **Nested loop** કહીયે છીએ.
- Nested loop કોઈ પણ પ્રકારની loop ની મદદથી બનાવી શકાય છે, પણ સૌથી વધારે for loop નો યુઝ કરીને nested loop નો ઉપયોગ કરવામાં આવે છે.

Control Flow



- If a loop exists inside the body of another loop, it's called a nested loop.

Syntax

```
for variable in sequence:
        # Outer Loop Statements

        for variable in sequence:
                # Inner Loop Statements

        # Outer Loop Statements
```

Example

```
for i in range(1, 6):
    print(f"Outer: {i}")
    for j in range(1, 3):
        print(f"i=>{i} & j=>{j}")
```

Output

```
Outer: 1
i=>1 & j=>1
i=>1 & j=>2
Outer: 2
i=>2 & j=>1
i=>2 & j=>2
Outer: 3
i=>3 & j=>1
i=>3 & j=>2
Outer: 4
i=>4 & j=>1
i=>4 & j=>2
Outer: 5
i=>5 & j=>1
i=>5 & j=>2
```

Example

```
for i in range(1, 6):
    for j in range(1, i+1):
        print(j, end=" ")
    print()
```

Output

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

LIST COMPREHENSION

- **List comprehension** એ કોઈ already બનેલ list પરથી નવી list બનાવવા માટેની રીત છે.
- List comprehension એ સામાન્ય રીતે કોઈ સામાન્ય functions અથવા loops કરતા વધારે compact અને fast હોય છે.
- છતાં પણ આપણે જેમ બને તેમ કોઈ લાબું અને complex coding ને list comprehension થી લખતા અટકવું જોઈએ જેથી આપણો code user-friendly રહી શકે.

NOTE:

- બધી જ list comprehension ને ફરીથી સરળ for loop ની મદદથી લખી શકાય છે. પણ બધી જ for loop ને list comprehension માં લખી શકાતું નથી.

Syntax

```
[ expression for item in sequence ]
```

Example

```
letters = [ i for i in 'python' ]
print( letters)
```

Output

```
[ ‘p’, ‘y’, ‘t’, ‘h’, ‘o’, ‘n’ ]
```

- List comprehensions ને કોઈ sequence ને કોઈ ચોક્કસ condition થી modify કરવા માટે પણ ઉપયોગમાં લેવામાં આવે છે.

Example

```
even_numbers = [ i for i in range(20) if i % 2 == 0]
print(even_numbers)
```

Output

```
[ 0, 2, 4, 6, 8, 10, 12, 14, 16, 18 ]
```

#4 FUNCTIONS & ARRAY IN DETAIL

BUILT-IN FUNCTIONS

- પાયથનમાં ઘણાંબધાં built-in functions આવે છે જેનો ઉપયોગ આપણે common tasks કરવા માટે વારંવાર કરતા હોઈએ છીએ.
- હાલમાં પાયથન total **68** built-in functions ધરાવે છે. તેમાંના અમુક વારંવાર ઉપયોગમાં આવતા functions નીચે જણાવેલ છે:

Function	Description
bin()	Returns the binary version of a number
chr()	Returns a character from the specified Unicode code.
dir()	Returns a list of the specified object's properties and methods
eval()	Evaluates and executes an expression
format()	Formats a specified value
help()	Executes the built-in help system
hex()	Converts a number into a hexadecimal value
id()	Returns the id of an object
input()	Allowing user input
len()	Returns the length of an object
map()	Returns the specified iterator with the specified function applied to each item
max()	Returns the largest item in an iterable
min()	Returns the smallest item in an iterable
oct()	Converts a number into an octal
open()	Opens a file and returns a file object
ord()	Convert an integer representing the Unicode of the specified character
pow()	Returns the value of x to the power of y
repr()	Returns a readable version of an object
reversed()	Returns a reversed iterator
sorted()	Returns a sorted list

@staticmethod()	Converts a method into a static method
sum()	Sums the items of an iterator
type()	Returns the type of an object

USER DEFINED FUNCTION (UDF)

- **UDFs** એવા functions હોય છે જે કોઈ નિચ્છીત કામ કરવા માટે **યુઝર દ્વારા** જ બનાવવામાં આવે છે. આવા functions code ના **re-use** માટે બનાવવામાં આવે છે.

Types of UDF:

A. Take Nothing, Return Nothing (**TNRN**)
B. Take Something, Return Nothing (**TSRN**)
C. Take Nothing, Return Something (**TNRS**)
D. Take Something, Return Something (**TSRS**)

Syntax - TNRN

Defining an UDF:
```
def functionName():
----# Function body
```

Calling an UDF:
```
functionName()
```

Syntax - TSRN

Defining an UDF:
```
def functionName( parameters ):
----# Function body
```

Calling an UDF:
```
functionName( arguments )
```

Syntax - TNRS

Defining an UDF:
```
def functionName():
----# Function body
----return something
```

Calling an UDF:
```
variable_name = functionName()
```

Syntax - TSRS

Defining an UDF:
```
def functionName( params ):
----# Function body
----return something
```

Calling an UDF:
```
variable_name = functionName( args )
```

Example - TNRN

```
def greetings():
    print("Good Day")

greetings()
```

Output

```
Good Day
```

Example - TSRN

```
def greetings(msg):
    print(msg)

greetings("Good Day")
```

Output

```
Good Day
```

Example - TNRS

```
def greetings():
    return "Good Day"

msg = greetings()
print(msg)
```

Output

```
Good Day
```

Example - TSRS

```
def greetings(msg):
    return msg

message = greetings("Good Day")
print(message)
```

Output

```
Good Day
```

TYPES OF FUNCTION ARGUMENTS

- When defining a function, we can catch arguments in following **3** ways:
  - **A.** Simple Arguments
  - **B.** Arbitrary Arguments
  - **C.** Keyword Arguments
  - **D.** Arbitrary Keyword Arguments
  - **E.** Default Parameter Value

A. Simple Arguments

- To specify simple parameters, we generally place variables in function definition to catch arguments.

Syntax

```
def functionName(arg1, arg2, ..., argN):
----# function body
```

Example

```
def test_param(n1, s1):
    print(n1)
    print(s1)

test_param(123, 456)
```

Output

```
123
456
```

B. Arbitrary Arguments

- If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.
- This way the function will receive a **tuple** of arguments, and can access the items accordingly.
- Arbitrary Arguments are often shortened to ***args** in Python documentations.

Syntax

```
def functionName(*args):
----# function body
```

Example

```
def courses(*args):
    print(arg[0])
    print(arg[2])

courses("GIM", "Web", "Flutter")
```

Output

```
GIM
Flutter
```

C. Keyword Arguments

- You can also send arguments with the **key = value** syntax.
- This way the **order** of the arguments **does not matter**.
- The phrase Keyword Arguments are often shortened to **kwargs** in Python documentations.

Syntax

```
Defining:
def functionName( param1, …, paramN ):
----# Function body

Calling:
functionName( param1=val1, …, paramN=valN )
```

Example

```
def test_param(n1, s1):
    print(n1)
    print(s1)

test_param(s1=123, n1=456)
```

Output

```
456
123
```

D. Arbitrary Keyword Arguments

- If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.
- This way the function will receive a **dictionary** of arguments, and can access the items accordingly.
- Arbitrary Keyword Arguments are often shortened to **kwargs in Python documentations.

Syntax

```
def functionName(**kwargs):
----# function body
```

Example

```
def about(**kwargs):
    print(f"Name: {kwargs['name']}")
    print(f"Age: {kwargs['age']}")

about(name="RNW", age="12")
```

Output

```
Name: RNW
Age: 12
```

E. Default Parameter Value

- The following example shows how to use a default parameter value.
- If we call the function without argument, it uses the default value.

Syntax

```
def functionName(param=val):
----# function body
```

Example

```
def country(name="India"):
    print(f"I love {name}")

country ("Ameria")
country()
```

Output

```
I love America
I love India
```

DOCUMENT STRING

- Python **documentation strings** (or **docstrings**) provide a convenient way of **associating documentation** with Python modules, functions, classes, and methods.
- It's specified in source code that is used, like a comment, to document a specific segment of code.
- Unlike conventional source code comments, the **docstring** should **describe what the function does**, not how.
- The docstrings are declared using **"""triple double quotes"""** just below the class, method or function declaration. All functions should have a docstring.
- The docstrings can be accessed using the **__doc__** method of the object or using the help function.

Syntax

```
def functionName():
----"""Document String."""
----# function body
```

Example

```
def country(name="India"):
    """This function prints out provided country name.

    If there is no any country name provided while
    calling a function, a default one which is India sis set
    to parameter.
    """
    print(f"I love {name}")

country()
print(country.__doc__)
```

Output

```
I love India
This function prints out provided country name.

If there is no any country name provided while calling a
function, a default one which is India sis set to parameter.
```

RECURSION

- Recursion means **function call itself**. This has the benefit of meaning that you can loop through data to reach a result.
- The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power.
- However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

Example

```
def fact(num):
    if num<=1:
        return 1
    else:
        return num*fact(num-1)

print(fact(4))
print(fact(5))
```

Output

```
24
120
```

ANONYMOUS / LAMBDA FUNCTION

- A function without a name is called as **Anonymous** or **Lambda function**.
- Use lambda functions when an anonymous function is required for a **short period of time**.
- A lambda function can take **any number of arguments**, but can only have **one expression**.

Syntax

```
lambda arguments : expression
```

Example

```
sqr = lambda n : n**2
print(sqr(3))

sumof = lambda a, b : a+b
print(sumof(3, 4))
```

Output

```
9
7
```

- The power of lambda is better shown when you use them as an **anonymous function inside another function**.
- Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number.

Example

```
def myFunc(num):
    return lambda a : a*num

mydoubler = myFunc(2)
mydoubler = myFunc(3)

print(mydoubler(11))
print(mydoubler(11))
```

Output

```
22
33
```

global KEYWORD

- The **global** keyword is used to create **global variables** from a no-global scope, e.g. inside a function.

Example

```
RNW = 5

def myFunc():
    global RNW
    RNW = 7
    print(RNW)

print(RNW)
myFunc()
print(RNW)
```

Output

```
5
7
7
```

RETURN MULTIPLE VALUES

- In Python, we can return multiple values from a function. Following are different ways:
  - **A.** Using Tuple
  - **B.** Using List
  - **C.** Using Dictionary

A. Using Tuple

- A Tuple is a comma separated sequence of items created with or without round brackets ( ).

Example

```
def myFunc():
    RNW = "Red & White"
    GIM = "Graduate in Multimedia"
    return RNW, GIM

name, course = myFunc()          # unpacking a tuple
print(name)
print(course)
```

Output

```
Red & White
Graduate in Multimedia
```

B. Using List

- A list is like an array of items created using square brackets [ ].

Example

```
def myFunc():
    RNW = "Red & White"
    GIM = "Graduate in Multimedia"
    return [ RNW, GIM ]

name, course = myFunc()          # unpacking a list
print(name)
print(course)
```

Output

```
Red & White
Graduate in Multimedia
```

C. Using Dictionary

- A dictionary is collection of items with key-value pair created using square brackets { }.

Example

```
def myFunc():
    d = dict()
    d['RNW'] = "Red & White"
    d['GIM'] = "Graduate in Multimedia"
    return d

about = myFunc()
print(about['RNW])
print(about['GIM'])
```

Output

```
Red & White
Graduate in Multimedia
```

ARRAY

- An **array** is a special variable, which can hold **more than one value at a time**. But all values must have to be a **same type**.
- Python does not have built-in support for Arrays, but Python **Lists** can be used instead.
- Following are different types of Arrays:
  A. 1D Array
  B. 2D Array

A. 1D Array

- An array which have **one-dimension** called as **1D Array**.

Syntax

```
variable = [ item1, item2, ..., itemN ]
```

Example

```
evens = [ 0, 2, 6, 12, 8, 4 ]
print(evens)                    # print whole array

print(evens[2])                 # print element on 2nd index

print(len(evens))               # prints length of an array

evens.append(16)                # add new element 16 at last

evens.pop(3)                    # remove an element from 3rd index

print(evens)
```

Output

```
[ 0, 2, 6, 12, 8, 4 ]
6
6
[ 0, 2, 6, 8, 4, 16]
```

B. 2D Array

- An array which have **two-dimension** called as **2D Array**. We can create a 2D array with help of **nested list**.

Syntax

```
variable = [ List1, List2, ..., ListN ]
```

Example

```
matrix = [ [ 3, 7, 1 ], [ 5, 2, 8 ], [ 9, 1, 5 ] ]

print(matrix)

print(matrix[1])

print(matrix[2][0])

for i in matrix:                    # matrix is 2D Array
    for j in i:                     # i becomes 1D Array
        print(j, end=" ")
    print()
```

Output

```
[ [ 3, 7, 1 ], [ 5, 2, 8 ], [ 9, 1, 5 ] ]
[ 5, 2, 8 ]
9
3 7 1
5 2 8
9 1 5
```

SORTING OF COLLECTION DATATYPES

- Sorting of Collection datatype can be applicable on List.
- To do sorting, we have two ways:
    - Using sort() method of List
    - Using sorted() built-in function

A. Using sort() method of list

- List items can be sorted using its **sort()** method. But using sort() we only sort it in **ascending order**.

Example

```
evens = [ 0, 2, 6, 12, 8, 4 ]
evens.sort()
print(evens)
```

Output

```
[ 0, 2, 4, 6, 8, 12 ]
```

B. Using sorted() method

- List items can be sorted using pythons built-in **sorted()** method. Using **sorted()** we can sort it in both **ascending order** and **descending order**.

Example

```
evens = [ 0, 2, 6, 12, 8, 4 ]
asc = sorted(evens)
print(asc)

desc = sorted(evens, reverse=True)
print(desc)
```

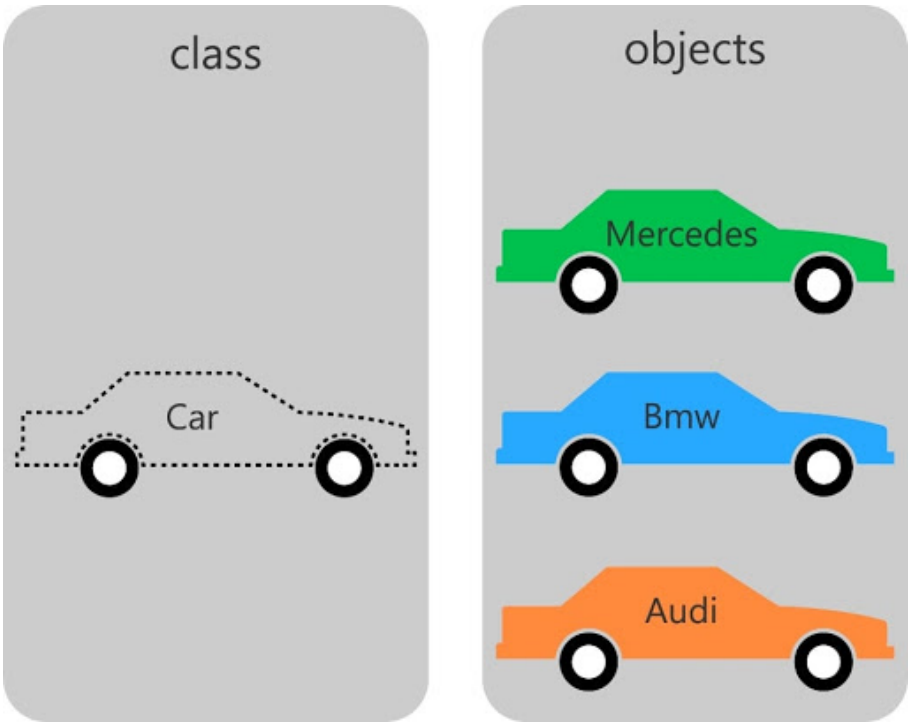
Output

```
[ 0, 2, 4, 6, 8, 12 ]
[ 12, 8, 6, 4, 2, 0 ]
```

#5 OBJECT ORIENTED PROGRAMMING (OOP)

CLASS & OBJECT

- A **class** is a group of objects which have common properties.
- It is a template or blueprint from which objects are created.
- It is a logical entity. It can't be physical.
- A class in Dart can contain:
  - Fields / Properties
  - Member Functions / Methods
  - Constructor
  - Destructor



- An entity that has state and behaviour is known as **object**.
- E.g., chair, bike, marker, pen, table, car, etc.
- An **object** is an **instance of a class**.

Syntax

Syntax to declare a class:

```
class ClassName:
    """
        Properties
        Methods
        Constructor
        Destructor
    """
```

Syntax to create an object:

```
object_name = ClassName()
```

Example

```
class Student:
    id = None                   # declaring a variable
    name = None                 # declaring a variable

s1 = Student()
print(s1.id)                    # accessing member through object
print(s1.name)                  # accessing member through object
```

Output

```
None
None
```

- In Python, "**Everything is an object**".
- So by default set object's initial value to **None**.

self & del KEYWORD

- The **self** parameter is a reference to the **current instance of the class**, and is used to access variables that belongs to the class.
- It does not have to be named self, you can call it whatever you like, but it **has to be the first parameter of any function in the class**.
- The self keyword is mainly used to eliminate the ambiguity between class attributes and parameters with the same name.

Example

```
class Student:
    id = None
    name = None

    def saveName(self, name):
        self.name = name

    def status(self):
        print("Enrolled")

s1 = Student()
print(s1.id)
print(s1.name)
s1.status()
s1.saveName("Rohan")
print(s1.name)
```

Output

```
None
None
Enrolled
Rohan
```

- You can **delete properties** on objects by using the **del** keyword. You can **delete objects** by using the **del** keyword.

Syntax

Syntax to delete a proprty:
del object.property

Syntax to delete an object:
del object

Example

```
del s1.name

del s1
```

ENCAPSULATION

- In an object oriented python program, you can **restrict access to methods and variables**. This can prevent the data from being modified by accident and is known as encapsulation.
- **Encapsulation** gives you more control over the degree of coupling in your code, it allows a class to change its implementation without affecting other parts of the code.
- Unlike other programming languages, Python don't have access specifiers like **public, private** and **protected**. But except protected in python, we can achieve same thing as public and private attributes using some other pythonic way.
- In Python, by default each attributes are **public**. And if we want to make it **private**, we just have to append prefix of **double underscore (__)** in front of that attribute or method.
- There is no support for **protected** as of now in Python.

Type	Description
public variables	Accessible from anywhere
public methods	Accessible from anywhere
private variables	Accessible only in their own class. Starts with **two underscores __**
private methods	Accessible only in their own class. Starts with **two underscores __**

Example

```
class Employee:
    id = None                   # public variable
    name = None                 # public variable
    __salary = None             # private variable

    def status(self):
        print("working")

    def __income(self):         # private method
        print(f"Salary: {self.__salary}")

e1 = Student()
print(e1.id, e1.name)
# print(e1.__salary)            # cannot accessed and raise an error
e1.status()
# e1.__income()                 # cannot accessed and raise an error
```

Output

```
None None
working
```

- Encapsulation prevents from accessing accidentally, but not intentionally.
- The private attributes and methods are **not really hidden**, they're renamed adding **_ClassName** in the beginning of their name.
- The method can actually be called using **object._ClassName__privateMethod()**

Example

```
class Employee:
    id = None
    name = None
    __salary = None

    def status(self):
        print("working")

    def __income(self):
        print(f"Salary: {self.__salary}")

e1 = Student()
print(e1.id, e1.name)
print(e1._Employee__salary)
e1.status()
e1._Employee__income()
```

Output

```
None None
None
Working
Salary: None
```

- Variables can be private which can be useful on many occasions. **A private variable can only be changed within a class method and not outside of the class**.
- Objects can hold crucial data for your application and you do not want that data to be changeable from anywhere in the code.

- If you want to change the value of a private variable, a **setter method** is used. This is simply a method that sets the value of a private variable. Same for getting all values, we should create a **getter method**.

Example

```
class Student:
    id = None
    name = None
    __contact = None

    def setData(self, id, name, contact):
        self.id = id
        self.name = name
        self.__contact = contact

    def getData(self):
        print(f"Id: {self.id}")
        print(f"Name: {self.name}")
        print(f"Contact: {self.__contact}")

s1 = Student()
s1.setData(1, "Rohan", 78945)
s1.getData()
```

Output

```
Id: Rohan
Name: Rohan
Contact: 78945
```

NOTE:

- As of global convention, we **always** should **create setter and getter method** (no matter what) to achieve full potential of encapsulation.

CONSTRUCTOR & DESTRUCTOR

- **Constructor** is a special **method that executes** when we **create an object** of a class.
- Constructor is defined using **__init__()** keyword.

Example

```
class City:

    def __init__(self):
        print("Object is created...")
        print("Welcome to my city.")

c1 = City()
c2 = City()
```

Output

```
Object is created...
Welcome to my city.
Object is created...
Welcome to my city.
```

- Constructor is generally used for **initialising class variables**.

Example

```
class Student:

    id = None
    name = None

    def __init__(self, id, name):
        self.id = id
        self.name = name

s1 = Student(1, "Red")
s2 = Student(2, "White")
print(s1.id, s1.name)
print(s2.id, s2.name)
```

Output

```
1 Red
2 White
```

NOTE:

- **If we initializing class variables through constructor, there is no need to declare a variable explicitly.**

Example

```
class Student:

    def __init__(self, id, name):
        self.id = id
        self.name = name

s1 = Student(1, "Red")
s2 = Student(2, "White")
print(s1.id, s1.name)
print(s2.id, s2.name)
```

Output

```
1 Red
2 White
```

- **Destructors** are called when an **object gets destroyed**. It's the polar opposite of the constructor, which gets called on creation.
- Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.
- Destructor is defined using **__del__()** keyword.

Example

```
class Student:

    def __init__(self, id, name):
        self.id = id
        self.name = name

    def __del__(self):
        print("Study Completed.")

s1 = Student(1, "Red")
s2 = Student(2, "White")
print(s1.id, s1.name)
print(s2.id, s2.name)
```

Output

```
1 Red
2 White
Study Completed.
Study Completed.
```

NESTED FUNCTION

- A function which is defined inside another function is known as **Nested Function**.
- Nested functions are able to access variables of the enclosing scope.
- In Python, these non-local variables can be accessed only within their scope and not outside their scope.

Example

```
def outer(n):
    msg = n
    def inner():
        print(msg)
    inner()                          # must be called inside this scope

outer("RNW")
# inner()                          # cannot be called
```

Output

```
RNW
```

- As we can see inner() can easily be accessed inside the outer body but not outside of its body. Hence, here, **inner()** is treated as **Nested function** which uses **msg** as non-local variable.

REFLECTION ENABLING FUNCTIONS

- **Reflection** refers to the **ability for code to be able to examine attributes** about objects that might be passed as parameters to a function.
- For example, if we write **type(obj)** then Python will return an object which represents the type of obj.
- Python has some reflection-enabling functions as following:
    - type()
    - isinstance()
    - callable()
    - dir()
    - getattr()

A. type()

- **type()** function takes one argument and **returns type** of given object.

Example

```
R = 5
W = "White"
L = [ 3, 8, 4 ]

print(type(R))
print(type(W))
print(type(L))
```

Output

```
<class 'int'>
<class 'str'>
<class 'list'>
```

B. isinstance()

- **isinstance()** function checks if the given object is an instance of given class or given tuple of classes and types.

Syntax

```
isinstance( object, class )

OR

isinstance( object, tuple_of_types )
```

Example

```
class Sample:
    pass

s1 = Sample()
print( isinstance(s1, Sample) )
print( isinstance(s1, (list, tuple)) )
print( isinstance(s1, (list, Sample)) )
```

Output

```
True
False
True
```

C. callable()

- A callable means anything that **can be called**.
- For an object, determines whether it can be called.
- The **callable()** method returns **True** if the object passed appears callable. If not, it returns **False**.

Example

```
RNW = 100

def about():
    print("Red & White")

print(callable(RNW))
print(callable(about))
```

Output

```
False
True
```

- A class can be made callable by providing a **__call__()** method.

Example

```
class Sample:

    def __call__(self):
        print("Sample is called...")

print(callable(Sample))
s1 = Sample()
print(s1.__call__())
```

Output

```
True
Sample is called...
```

D. dir()

- The **dir()** method tries to return a **list of valid attributes** of the object.

Example

```
RNW = [ “GIM”, “Flutter”, “Python” ]

print(dir(RNW))
```

Output

```
[ '__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__',
'__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort' ]
```

E. getattr()

- The **getattr()** method returns the value of the named attribute of an object.
- If not found, it returns the default value provided to the function.
- The getattr() method **takes three parameters** object, name and default(optional).

Syntax

```
getattr( object, “attribute_name” )

OR

isinstance( object, “attribute_name”, default_value)
```

Example

```
class Student:
    name = "Rohan"
    age = 18

s1 = Student()
print("Name: ", getattr(s1, "name"))
print("Age: ", getattr(s1, "age"))
print("Contact: ", getattr(s1, "contact", 78945))
```

Output

```
Name: Rohan
Age: 18
Contact: 78945
```

TYPES OF INHERITANCE

- **Inheritance** allows us to define a class that inherits all the methods and properties from another class.
- In Inheritance, we have two types of class basically:
    - Parent Class
    - Child Class

Parent Class:

- This is the class which properties or characteristics being inherited by other class.
- **Parent class** is also called **Super class** or **Base class**.

Child Class:

- This is the class which inherit some other class's properties or characteristics.
- **Child class** is also called **Sub class** or **Derived class**.

Syntax

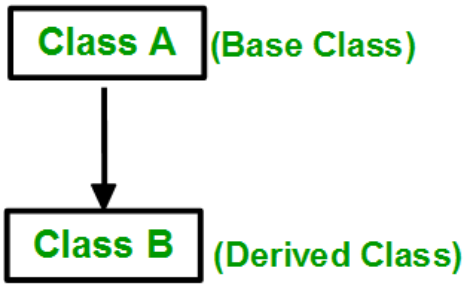
```
class child_class( parent_class ):
    pass
```

Types of Inheritance:

- Python provides 5 types of inheritance:
  - **A. Single Level Inheritance**
  - **B. Multi-level Inheritance**
  - **C. Multiple Inheritance**
  - **D. Hierarchical Inheritance**
  - **E. Hybrid Inheritance**

A. Single Level Inheritance

- In Single Inheritance, only one class can be inherit from another class. That means, only **one parent class** and **one child class**.



Example

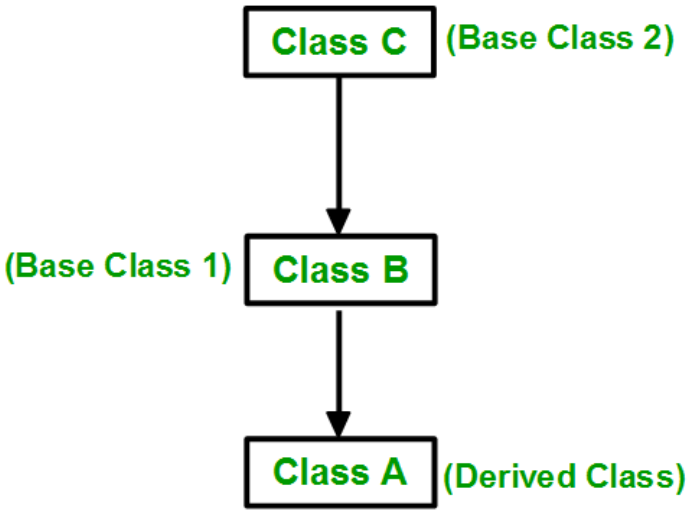
```
class Animal:
    def showName(self, name):
        print(name)

    def showAge(self, age):
        print(age)

class Lion(Animal):
    pass


l1 = Lion()
l1.showName("Jack")
l1.showAge(20)
```

Output

```
Jack
20
```

B. Multi-level Inheritance

- In Multi-level Inheritance, a **child class has its own child class**.



Example

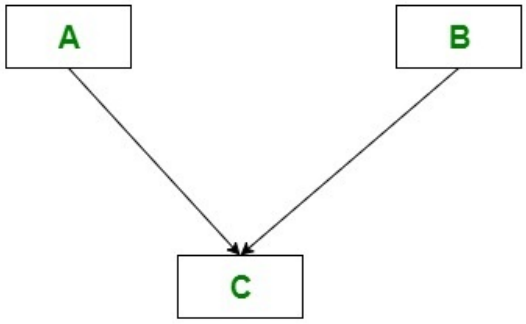
```
class Grandfather:
    def showAge(self, name):
        print(name)

class Father(Grandfather):
    def showProfession(self, profession):
        print(profession)

class Son(Father):
    pass

s1 = Son()
s1.showAge(23)
s1.showProfession("Engineer")
```

Output

```
23
Engineer
```

C. Multiple Inheritance

- In Multiple Inheritance, a **single child class** can have **two or more parent class**.



Example

```
class A:
    def skill(self):
        print("driving.")

class B:
    def skill(self):
        print("swimming.")

    def ability(self):
        print("lots of books reading quickly.")

class C(A, B):
    pass

c1 = C()
c1.skill()
c1.ability()
```

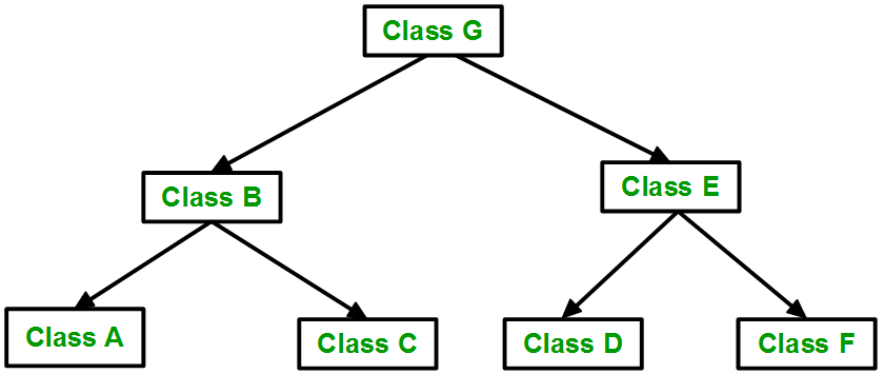
Output

```
driving.
lots of books reading quickly.
```

- In above example, skill() method is called from class A because while specifying child class, we write class C(**A**, B) where we put class **A** first.

D. Hierarchical Inheritance

- In Hierarchical Inheritance, **many child class** can be created from **one parent class**. And from that child class, even further child class can be created.



Example

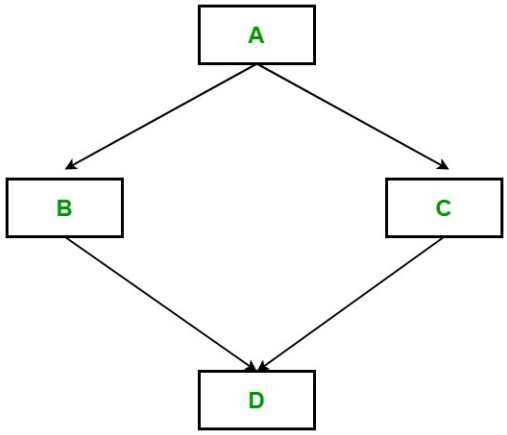
```
class A:
    def setter(self):
        self.x = 3

class B(A):
    def disp(self):
        print(self.x)

class C(A):
    def shower(self):
        self.x = 9
        print(self.x)


b1 = B()
b1.setter()
b1.disp()

c1 = C()
c1.setter()
c1.shower()
```

Output

```
3
9
```

E. Hybrid Inheritance

- In Hybrid Inheritance, **many child class** can be created from **one or more parent class**. And from that child class, even further child class can be created.
- In other words, **combination of any two or more inheritance** types is called Hybrid inheritance.



Example

```
class A:
    def setter(self):
        self.x = 3

class B(A):
    def disp(self):
        print(self.x)
    def getter(self):
        print(f"From class B: x={self.x}")

class C(A):
    def shower(self):
        self.x = 9
        print(self.x)
    def getter(self):
        print(f"From class C: x={self.x}")

class D(B, C):
    pass

d1 = D()
d1.setter()
d1.disp()
d1.getter()
d1.shower()
d1.getter()
```

Output

```
3
From class B: x=3
9
From class B: x=9
```

METHOD OVERLOADING & METHOD OVERRIDING

- Polymorphism means the ability to take various forms.
- We can achieve polymorphism by following two approaches:
    - Method Overloading
    - Method Overriding

A. Method Overloading

- Method Overloading is a feature that allows a class to have **more than one method** having the **same name**, if their argument lists are different.
- Unfortunately, Method overloading not supported in Python.

B. Method Overriding

- Declaring **a method in sub class** which **is already present in parent class** is known as **method overriding**.
- Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class.
- In this case the method in **parent class** is called **overridden method** and the method in **child class** is called **overriding method**.
- Both overridden and overriding method must have common structure(same arguments) to achieve method overriding.
- The main advantage of method overriding is that the class can give its own specific implementation to an inherited method **without even modifying the parent class code**.

Example

```
class Car:

    def ShowOwner():
        print("This is Parent's car")


class Jay(Car):

    def ShowOwner():
        print("This JD's car")


class Sanket(Car):

    def ShowOwner():
        print("This is Sanket's car")


parentCar = Car()
jdCar = Jay()
sanketCar = Sanket()

parentCar.ShowOwner()
jdCar.ShowOwner()
sanketCar.ShowOwner()
```

Output

```
This is Parent's car
This JD's car
This is Sanket's car
```

issubclass() & super() METHODS

- The **issubclass()** function returns **True** if the specified object is a subclass of the specified object, otherwise **False**.

Syntax

```
issubclass(object, subclass)
```

Example

```
class Vehicle:
    pass

class Car(Vehicle):
    pass

print( issubclass(Car, Vehicle) )
```

Output

```
True
```

- The **super()** function in Python makes class inheritance more manageable and extensible.
- The function **returns a temporary object** that allows **reference to a parent class** by the keyword super.
- **super() function will make the child class inherit all the methods and properties from its parent**.
- The super() function has two major use cases:
    - To avoid the usage of the super (parent) class explicitly.
    - To enable multiple inheritances.

A. To avoid the usage of the super (parent) class explicitly

- Suppose we have created a child class that inherits the properties and methods from its parent.
- We want to add the __init__() function to the child class.
- When you add the **__init__()** function, the **child class will no longer inherit the parent's __init__() function**.

NOTE:

- **The child's __init__() function overrides the inheritance of the parent's __init__() function.**

- **To keep** the inheritance of the parent's __init__() function, add a **call to the parent's __init__()** function.

Example

```
class Vehicle:
    def __init__(self):
        print("From parent class.")

class Car(Vehicle):
    def __init__(self):
        print("From child class.")
        Vehicle.__init__(self)

c1 = Car()
```

Output

```
From child class.
From parent class.
```

- By using the **super()** function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

Example

```
class Vehicle:
    def __init__(self):
        print("From parent class.")

class Car(Vehicle):
    def __init__(self):
        print("From child class.")
        super().__init__()

c1 = Car()
```

Output

```
From child class.
From parent class.
```

BUILT-IN DUNDER METHODS

- In Python, **special methods** are a set of **predefined methods** you can use to enrich your classes. They are easy to recognize because they **start and end with double underscores**, for example **__init__** or **__str__**.
- These "dunders" or "special methods" in Python are also sometimes called "**magic methods**". But using this terminology can make them seem more complicated than they really are—at the end of the day there's nothing "magical" about them. You should treat these methods like a normal language feature.
- Using dunder methods, we can achieve or enhance following situations:
    - Object Initialization
    - Object Representation
    - Enable Iteration
    - Callable Objects
    - Operator Overloading

A. Object Initialization

- Object can be initialized using the constructor **__init__** dunder method.

Example

```
class Account:
    def __init__(self, owner, amount=0):
        self.owner = owner
        self.amount = amount
        self.transactions = []

acc1 = Account("Rohan")
acc2 = Account("Dravid", 1000)
```

B. Object Representation

- It's common practice in Python to provide a string representation of your object for the consumer of your class (a bit like API documentation.) There are two ways to do this using dunder methods:
    - **__repr__:** The "**official**" string representation of an object. This is how you would make an object of the class. The goal of __repr__ is to be unambiguous.
    - **__str__:** The "**informal**" or nicely printable string representation of an object. This is for the **enduser**.
- Let's implement these two methods on the Account class:

Example

```
class Account:
    def __init__(self, owner, amount=0):
        self.owner = owner
        self.amount = amount
        self.transactions = []

    def __repr__(self):
        return f"Account({self.owner}, {self.amount})"

    def __str__(self):
        return f"Account of {self.owner} is {self.amount}"

acc1 = Account("Rohan")
acc2 = Account("Dravid", 1000)
```

- If you don't want to hardcode "Account" as the name for the class you can also use **self.__class__.__name__** to access it programmatically.

NOTE:
- **If you wanted to implement just one of these to-string methods on a Python class, make sure it's __repr__.**

- Now we can query the object in various ways and always get a nice string representation:

Example

```
print(acc1)
print( str(acc1) )
print( repr(acc2) )
```

Output

```
Account of Rohan is 0
Account of Rohan is 0
Account(Rohan, 1000)
```

C. Enable Iteration

- In order to iterate over our account object we need to add some transactions. So first, we'll define a simple method to add transactions.

Example

```
def add_transaction(self, amount):
    if amount is int:
        self.transactions.append(amount)
```

- We also defined a method to calculate the balance on the account so we can conveniently access it with account.balance.
- This method takes the start amount and adds a sum of all the transactions:

Example

```
def balance (self):
    print(self.amount + sum(self.transactions))
```

- Let's do some deposits and withdrawals on the account:

Example

```
acc3 = Account("Raj", 2000)
acc3.add_transaction(500)
acc3.add_transaction(300)
acc3.add_transaction(-100)
acc3.add_transaction(200)
acc3.balance()
```

Output

```
2900
```

- Now we have some data and we want to know:
  1. How many transactions were there?
  2. Index the account object to get transaction number
  3. Loop over the transactions
- With the class definition we have this is currently not possible. All of the following statements raise TypeError exceptions:

Example

```
>>> len(acc3)
TypeError

>>> for bal in acc3:
        print(bal)
TypeError

>>> acc3[1]
TypeError
```

- Dunder methods to the rescue! It only takes a little bit of code to make the class iterable:

Example

```
def __len__(self):
    return len(self.transactions)

def __getitem__(self, position):
    return self.transactions[position]
```

- Now the previous statements work:

Example

```
>>> len(acc3)
4

>>> for bal in acc3:
        print(bal)
500
300
-100
200

>>> acc3[1]
300
```

- To iterate over transactions in reversed order you can implement the **__reversed__** special method:

Example

```
def __reversed__(self):
    return self[ : : -1]

>>> list( reversed(acc3) )
[ 200, -100, 300, 500 ]
```

D. Callable Objects

- We can make an object callable like a regular function by adding the **__call__** dunder method.
- For our account class we could print a nice report of all the transactions that make up its balance:

Example

```
def __call__(self):
    print(f"Initial amount: {self.amount}")
    print("Transactions:")
    for trans in self:
        print(trans)
    self.balance()
```

- Now when we call the object with the double-parentheses **acc()** syntax, we get a nice account statement with an overview of all transactions and the current balance:

Example

```
>>> acc = Account("Rohan", 2000)
>>> acc.add_transaction(500)
>>> acc.add_transaction(300)
>>> acc.add_transaction(-100)
>>> acc.add_transaction(200)

>>> acc()
Initial amount: 2000
Transactions:
500
300
-100
200
Balance: 2900
```

OPERATOR OVERLOADING

- In Python, everything is an object. We are completely fine adding two integers or two strings with the **+ (plus) operator**, it behaves in expected ways.
- But **+ operator** don't actually work with class objects:

Example

```
>>> class Complex:

            def set(self, n1, n2):
                self.n1 = n1
                self.n2 = n2

            def get(self):
                print("n1 = ", self.n1, " n2 = ", self.n2)


>>> c1 = Complex()
>>> c2 = Complex()
>>> c3 = Complex()

>>> c1.set(5, 4)
>>> c2.set(3, 1)

>>> c1.get()
n1 =  5  n2 =  4

>>> c2.get()
n1 =  3  n2 =  1

>>> c3 = c1 + c2
TypeError: unsupported operand type(s) for +: 'Complex' and 'Complex'
```

- We can solve this by implementing **__add__** dunder method.

Example

```
def __add__(self, obj):
    temp = Complex()
    temp.n1 = self.n1 + obj.n1
    temp.n2 = self.n2 + obj.n2
    return temp
```

- Now, we can use **+ operator** for adding two actual objects:

Example

```
>>> c1 = Complex()
>>> c2 = Complex()
>>> c3 = Complex()

>>> c1.set(5, 4)
>>> c2.set(3, 1)

>>> c1.get()
n1 =  5  n2 =  4

>>> c2.get()
n1 =  3  n2 =  1

>>> c3 = c1 + c2
>>> c3.get()
n1 =  8  n2 =  5
```

- Python magic methods for operator overloading:

Binary Operators:

Operators	Magic Methods
+	__add__(self, other)
-	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)
//	__floordiv__(self, other)

Unary Operators:

Operators	Magic Methods
+	__pos__(self, other)
-	__neg__(self, other)
~	__invert__(self, other)

Comparision Operators:

Operators	Magic Methods
<	__lt__(self, other)
>	__gt__(self, other)
<=	__le__(self, other)
>=	__ge__(self, other)
==	__eq__(self, other)
!=	__ne__(self, other)

Assignment Operators:

Operators	Magic Methods
+=	__iadd__(self, other)
-=	__isub__(self, other)
*=	__mul__(self, other)
/=	__idiv__(self, other)
%=	__imod__(self, other)
**=	__ipow__(self, other)
//=	__ifloordiv__(self, other)

#6 EXCEPTION HANDLING

try … except

- When an **error** occurs, or **exception** as we call it, Python will normally **stop and generate an error message**.

Example

```
>>> print(RNW)
NameError: name 'RNW' is not defined
```

- These exceptions can be handled using the **try** statement.
- Since the **try** block raises an error, the **except** block will be executed.
- Without the try block, the program will crash and raise an error.
- The **try** block lets you **test a block of code for errors**.
- The **except** block lets you **handle the error**.

Syntax

```
try:
----# code that may raise an error/exception
except:
----# a solution message instead of error
```

Example

```
try:
    print(RNW)
except:
    print("An exception occurred.")
```

Output

```
An exception occurred.
```

- You can define as **many exception blocks** as you want, e.g. if you want to execute a special block of code for a **special kind of error**.

Example

```
try:
    print(RNW)
except NameError:
    print("Variable is not defined yet.")
except:
    print("Something else went wrong.")
```

Output

```
Variable is not defined yet.
```

- You can also **store an exception message** into an **object** using **as** keyword. That object will be a type of raised exception.

Example

```
try:
    print(RNW)
except NameError as e:
    print("Not possible because", e)
except:
    print("Something else went wrong.")
```

Output

```
Not possible because name 'RNW' is not defined
```

try … except … else

- You can use the **else** keyword to define a block of code to be executed **if no errors were raised**.

Example

```
try:
    print("Hey, There!")
except:
    print("Exception is occurred.")
else:
    print("All is fine.")
```

Output

```
Hey, There!
All is fine.
```

try … except … finally

- The **finally** block, if specified, will be executed regardless **if the try block raises an error or not**.

Example

```
try:
    print(RNW)
except:
    print("Exception is occurred.")
finally:
    print("Always executes.")
```

Output

```
Exception is occurred.
Always executes.
```

try … except … else … finally

- The **finally** block can also be used with **else** block.
- If there is no any exception or error occurrs then both **else block** and **finally block** executes.

Example

```
try:
    print("Hello")
except:
    print("Exception is occurred.")
else:
    print("No Errors.")
finally:
    print("Always executes.")
```

Output

```
Hello
No Errors.
Always executes.
```

- If there is any exception or error occurrs then both **except block** and **finally block** executes.

Example

```
try:
    print(RNW)
except:
    print("Exception is occurred.")
else:
    print("No Errors.")
finally:
    print("Always executes.")
```

Output

```
Exception is occurred.
Always executes.
```

raise & assert KEYWORD

- As a Python developer you can choose to **throw an exception** if a condition occurs.
- To throw (or raise) an exception, use the **raise** keyword.

Example

```
RNW = 3

if RNW < 5:
      raise Exception(“Invalid Value.”)
```

Output

```
Traceback (most recent call last):
  File "<pyshell#8>", line 4, in <module>
    raise Exception("Invalid Value.")
Exception: Invalid Value.
```

- The **raise** keyword is used to raise an exception.
- You can define what kind of error to raise, and the text to print to the user.

Example

```
RNW = 3

if RNW is not str:
      raise TypeError(“Invalid Type.”)
```

Output

```
Traceback (most recent call last):
  File "<pyshell#8>", line 4, in <module>
    raise TypeError("Invalid Type.")
TypeError: Invalid Type.
```

- The **assert** keyword is used when **debugging code**.
- The **assert** keyword lets you **test if a condition in your code returns True**, if not, the program will raise an **AssertionError**.

Example

```
>>> quality = 80

>>> assert quality > 70          # if condition is True, then nothing happens

>>> assert quality > 90
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    assert quality>90
AssertionError
```

- You can write a **message** to be written if the code returns **False**, as shown in example below:

Example

```
>>> quality = 80

>>> assert quality > 90 , “Validation failed…”
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    assert quality>90 , “Validation failed…”
AssertionError: Validation failed…
```

CUSTOM EXCEPTION

- Python has many built-in exceptions which forces your program to output an error when something in it goes wrong.
- However, sometimes you may need to **create custom exceptions that serves your purpose**.
- In Python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from **Exception class**. Most of the built-in exceptions are also derived from this class.
- This new exception can be raised, like other exceptions, using the **raise** statement with an **optional error message**.

Example

```
class MyException(Exception):
    pass

raise MyException
```

Output

```
Traceback (most recent call last):
  File "<pyshell#8>", line 4, in <module>
    raise MyException
MyException
```

- You can also pass a message while raise an exception:

Example

```
class MyException(Exception):
    pass

raise MyException("My custom exception works...")
```

Output

```
Traceback (most recent call last):
  File "<pyshell#8>", line 4, in <module>
    raise MyException("My custom exception works...")
MyException: My custom exception works...
```

#7 FILE HANDLING

MODES OF OPENING FILE

- **File handling** is an important part of any web application.
- Python has several functions for creating, reading, updating, and deleting files.
- The key function for working with files in Python is the **open()** function.
- The open() function takes **two parameters; filename**, and **mode**.
- There are four different methods (modes) for opening a file:
    - r - **Read** - Default value. Opens a file for reading, error if the file does not exist
    - a - **Append** - Opens a file for appending, creates the file if it does not exist
    - w - **Write** - Opens a file for writing, creates the file if it does not exist
    - x - **Create** - Creates the specified file, returns an error if the file exists
    - + - Opens a file for updating (reading and writing)

- In addition you can specify if the file should be handled as **binary mode** or **text mode**:
    - t - **Text** - Default value. Text mode
    - b - **Binary** - Binary mode (e.g. images)

Syntax

```
open( filename, mode )
```

Example

```
f = open("RNW.txt")          # by default in read mode
f = open("C:\\Python38\\RNW.txt") # from specific directory
f = open("RNW.txt", "r")     # by default in text mode
f = open("RNW.txt", "rt")    # same as above
f = open("Red.png", "r+b")   # read and write in binary mode
```

I/O OPERATION WITH FILE

- When we want to **read** from or **write** to a file, we need to **open** it first.
- When we are done, it needs to be **closed** so that the resources that are tied with the file are freed.
- Hence, in Python, a file operation takes place in the following order:
    - Open a file
    - Read or write (perform operation)
    - Close the file

- We already see how to open a file. Let's see how to close a file.
- When we are done with performing operations on the file, we need to properly close the file.
- Closing a file will free up the resources that were tied with the file. It is done using the **close()** method available in Python.
- Python has a garbage collector to clean up unreferenced objects but we must not rely on it to close the file.

Example

```
f = open("RNW.txt")

# perform file operations

f.close()
```

- This method is not entirely safe. If an **exception** occurs when we are performing some operation with the file, the code exits without closing the file.
- A safer way is to use a **try...finally** block.

Example

```
try:
    f = open("RNW.txt")
    # perform file operations
finally:
    f.close()
```

- This way, we are guaranteeing that the file is properly closed even if an exception is raised that causes program flow to stop.

- The best way to close a file is by using the **with** statement. This ensures that the file is closed when the block inside the with statement is exited.
- **We don't need to explicitly call the close() method. It is done internally.**

Example

```
with open("RNW.txt") as f:
    # perform file operations
```

file object attributes:

- Once a file is opened and you have one file object, you can get various information related to that file.
- Here is a list of all attributes related to file object:

Attributes	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.

Example

```
with open("RNW.txt") as f:
    print(f.closed)
    print(f.mode)
    print(f.name)

print(f.closed)
```

Output

```
False
r
RNW.txt
True
```

Writing to a File:

- In order to write into a file in Python, we need to open it in **write w**, **append a or exclusive creation x mode**.
- We need to be careful with the **w** mode, as it will **overwrite into the file if it already exists**. Due to this, all the **previous data are erased**.
- Writing a string or sequence of bytes (for binary files) is done using the **write()** method. This method returns the number of characters written to the file.

Example

```
with open("RNW.txt", "w") as f:
    f.write("Red & White\n")
    f.write("Graduate in Multimedia\n")
    f.write("International\n")
```

- This program will create a new file named RNW.txt in the current directory if it does not exist. If it does exist, it is overwritten.
- We must include the newline characters ourselves to distinguish the different lines.

Reading a File:

- To **read** a file in Python, we must open the file in reading **r mode**.
- There are various methods available for this purpose. We can use the **read(size)** method to read in the size number of data.
- **If the size parameter is not specified, it reads and returns up to the end of the file**.
- We can read the RNW.txt file we wrote in the above section in the following way:

Example

```
>>> f = open("RNW.txt", "r")

>>> print(f.read(3))            # read the first 3 characters
'Red'

>>> print(f.read(3))            # read the next 3 characters
' & '

>>> print(f.read())             # read rest of the data till end of the file
'White\nGroup of Institute\nInternational\n'

>>> print(f.read())             # further reading returns empty string
''
```

- We can see that the **read()** method returns a newline as '\n'. Once the end of the file is reached, we get an empty string on further reading.
- We can **change our current file cursor** (position) using the **seek()** method.
- Similarly, the **tell()** method returns our **current position** (in number of bytes).

Example

```
>>> f.tell()              # get current file or cursor position
49

>>> f.seak(0)             # bring file cursor to initial position
0

>>> print(f.read())       # read the entire file
Red & White
Graduate in Multimedia
International
```

- We can read a file line-by-line using a **for loop**. This is both efficient and fast.

Example

```
>>> for line in f:
        print(line, end="")

Red & White
Graduate in Multimedia
International
```

- Alternatively, we can use the **readline()** method to read individual lines of a file. This method reads a file till the **newline**, including the newline character.

Example

```
>>> f.readline()
Red & White

>>> f.readline()
Graduate in Multimedia

>>> f.readline()
International
```

- Lastly, the **readlines()** method returns a list of remaining lines of the entire file. All these reading methods return empty values when the **end of file (EOF)** is reached.

Example

```
>>> f.readlines()
[ 'Red & White\n', 'Group of Institutes\n', 'International\n' ]
```

#8 WORKING WITH MODULES

datetime MODULE

- Python has a module named **datetime** to work with **dates and times**.
- To import any module in Python, use following syntax:

Syntax

```
import module_name
```

- We will see datetime module by exploring various examples:

Get Current Date and Time:

- One of the classes defined in the **datetime** module is **datetime class**.
- We then use **now()** method to create a datetime object containing the current local date and time.

Example

```
import datetime

dt = datetime.datetime.now()
print(dt)
```

Output

```
2020-05-06 09:00:24.584593
```

Get Current Date:

- One In this program, we use **today()** method defined in the **date class** to get a date object containing the current local date.

Example

```
import datetime

dt = datetime.date.today()
print(dt)
```

Output

```
2020-05-06
```

- We can use **dir()** method to get a list containing all attributes of a module.

Example

```
import datetime

print( dir(datetime) )
```

Output

```
['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__',
'__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'date', 'datetime',
'datetime_CAPI', 'sys', 'time', 'timedelta', 'timezone',
'tzinfo']
```

- Commonly used classes in the datetime module are:
    - date class
    - time class
    - datetime class
    - timedelta class

A. datetime.date class

- You can instantiate **date** objects from the **date** class.
- A date object represents a **date (year, month and day)**.

Date object to represent a date:

- **date()** in the below example is a constructor of the **date** class. The constructor takes three arguments: **year, month** and **day**.
- The variable dt is a date object.

Example

```
import datetime

dt = datetime.date(2020, 5, 15)
print(dt)
```

Output

```
2020-05-15
```

Get date from a timestamp:

- We can also create **date** objects from a **timestamp**.
- A Unix timestamp is the number of seconds between a particular date and **January 1, 1970** at UTC.
- You can convert a timestamp to date using **fromtimestamp()** method.

Example

```
import datetime

dt = datetime.date.fromtimestamp(1645789854)
print(dt)
```

Output

```
2022-02-25
```

Print today's year, month and day:

- We can get year, month, day, day of the week etc. from the date object easily.

Example

```
import datetime

dt = datetime.date.today()
print(dt.year)
print(dt.month)
print(dt.day)
```

Output

```
2020
5
6
```

B. datetime.time class

- A time object instantiated from the **time** class represents the local time.

Time object to represent time:

- **time()** in the below example is a constructor of the **time** class. The constructor takes four optional arguments: **hour, minute, second** and **microsecond**.

Example

```
import datetime

a = datetime.time()
print(a)

b = datetime.time(12, 25, 48)
print(b)

c = datetime.time(hour=12, minute=25, second=48)
print(c)

d = datetime.time(12, 25, 48, 215678)
print(d)
```

Output

```
00 : 00 : 00
12 : 25 : 48
12 : 25 : 48
12 : 25 : 48.215678
```

Print hour, minute, second and microsecond:

- Once you create a **time** object, you can easily print its attributes such as **hour, minute** etc.
- If we don't pass **microsecond** argument, then it will be by-default **0**.

Example

```
import datetime

a = datetime.time(12, 25, 48)

print("Hour: ", a.hour)
print("Minute: ", a.minute)
print("Second: ", a.second)
print("Microsecond: ", a.microsecond)
```

Output

```
Hour: 12
Minute: 25
Second: 48
Microsecond: 0
```

C. datetime.datetime class

- The datetime module has a class named **datetime** that can contain information from both **date** and **time** objects.

Python datetime object:

- **datetime()** in the below example is a constructor of the **datetime** class. The constructor takes total 7 arguments: **year, month, day, hour, minute, second** and **microsecond**.
- In which first three arguments are mandatory.

Example

```
import datetime

a = datetime.datetime(2020, 5, 15)
print(a)

b = datetime.datetime(2020, 5, 15, 6, 26, 45, 234589)
print(b)
```

Output

```
2020-05-15 00:00:00
2020-05-15 06:26:45.234589
```

Print year, month, hour, minute and timestamp:

- Once you create a **datetime** object, you can easily print its attributes such as **year, month** etc.

Example

```
import datetime

a = datetime.datetime(2020, 5, 15, 6, 26, 45, 234589)
print("Year: ", a.year)
print("Month: ", a.month)
print("Hour: ", a.hour)
print("Minute: ", a.minute)
print("Timestamp: ", a.timestamp())
```

Output

```
Year: 2020
Month: 5
Hour: 6
Minute: 26
Timestamp: 1589504205.234589
```

D. datetime.timedelta class

- A **timedelta** object represents the **difference** between **two dates or times**.

Difference between two dates and times:

- We can also import any class from datetime module individually using following syntax:

Syntax

```
from module import class
```

Example

```
from datetime import datetime, date

t1 = date(year = 2018, month = 7, day = 12)
t2 = date(year = 2017, month = 12, day = 23)
t3 = t1 - t2
print("t3 =", t3)

t4 = datetime(year = 2018, month = 7, day = 12, hour = 7, minute = 9, second = 33)
t5 = datetime(year = 2019, month = 6, day = 10, hour = 5, minute = 55, second = 13)
t6 = t4 - t5
print("t6 =", t6)

print("type of t3 =", type(t3))
print("type of t6 =", type(t6))
```

Output

```
t3 = 201 days, 0:00:00
t6 = -333 days, 1:14:20
type of t3 = <class 'datetime.timedelta'>
type of t6 = <class 'datetime.timedelta'>
```

Difference between two timedelta objects:

- We will create two timedelta objects t1 and t2, and find their difference.

Example

```
from datetime import timedelta

t1 = timedelta(weeks = 2, days = 5, hours = 1, seconds = 33)
t2 = timedelta(days = 4, hours = 11, minutes = 4, seconds = 54)
t3 = t1 - t2

print("t3 =", t3)
```

Output

```
t3 = 14 days, 13:55:39
```

Printing negative timedelta object:

- We will create two timedelta objects t1 and t2, and find their negative difference.

Example

```
from datetime import timedelta

t1 = timedelta(seconds = 45)
t2 = timedelta(seconds = 57)
t3 = t1 - t2

print("t3 =", t3)
print("t3 =", abs(t3))
```

Output

```
t3 = -1 day, 23:59:48
t3 = 0:00:12
```

Time duration in seconds:

- You can get the total number of seconds in a timedelta object using **total_seconds()** method.
- You can also find sum of two dates and times using **+** operator.
- Also, you can multiply and divide a **timedelta** object by integers and floats.

Example

```
from datetime import timedelta

t = timedelta(days = 3, hours = 2, seconds = 45, microseconds = 124589)
print("Total seconds =", t.total_seconds())
```

Output

```
Total seconds = 266445.124589
```

time MODULE

- Python has a module named **time** to handle time-related tasks.
- Here are commonly used time-related functions:

time.time():

- The **time()** function returns the **number of seconds passed since epoch**.
- For Unix system, **January 1, 1970, 00:00:00** at UTC is **epoch** (the point where time begins).

Example

```
import time

seconds = time.time()
print("Seconds since epoch =", seconds)
```

Output

```
Seconds since epoch = 1588748050.2725801
```

time.ctime():

- The **time.ctime()** function takes seconds passed since epoch as an argument and returns a **string representing local time**.

Example

```
import time

seconds = 1588748254.143967                    # seconds passed since epoch
local_time = time.ctime(seconds)
print("Local time:", local_time)
```

Output

```
Local time: Wed May  6 12:27:34 2020
```

time.sleep():

- The **sleep()** function **suspends** (delays) execution of the current thread for the given number of **seconds**.

Example

```
import time

print("Welcome to...")
time.sleep(2)
print("Red & White Group of Institutes")              # executes after 2 seconds
```

Output

```
Welcome to...
Red & White Group of Institutes
```

time.struct_time Class:

- Several functions in the time module such as **gmtime()**, **asctime()** etc. either take **time.struct_time** object as an argument or return it.
- Here's an example of **time.struct_time** object:

Example

```
time.struct_time(tm_year=2018, tm_mon=12, tm_mday=27,
                 tm_hour=6, tm_min=35, tm_sec=17,
                 tm_wday=3, tm_yday=361, tm_isdst=0)
```

- The values (elements) of the **time.struct_time** object are accessible using both **indices** and **attributes**:

Index	Attribute	Values
0	**tm_year**	0000, ..., 2018, ..., 9999
1	**tm_mon**	1, 2, ..., 12
2	**tm_mday**	1, 2, ..., 31
3	**tm_hour**	0, 1, ..., 23
4	**tm_min**	0, 1, ..., 59
5	**tm_sec**	0, 1, ..., 61
6	**tm_wday**	0, 1, ..., 6; Monday is 0
7	**tm_yday**	1, 2, ..., 366
8	**tm_isdst**	0, 1 or -1

time.localtime():

- The **localtime()** function takes the number of seconds passed since epoch as an argument and returns **struct_time** in **local time**.

Example

```
import time

result = time.localtime(1588748254)
print("Result:", result)
print("\nYear:", result.tm_year)
print("tm_hour:", result.tm_hour)
```

Output

```
Result: time.struct_time(tm_year=2020, tm_mon=5, tm_mday=6, tm_hour=12,
tm_min=27, tm_sec=34, tm_wday=2, tm_yday=127, tm_isdst=0)

Year: 2020
tm_hour: 12
```

time.strftime():

- The **strftime()** function takes **struct_time** (or **tuple** corresponding to it) as an argument and returns a **string** representing it based on the format code used.

Example

```
import time

named_tuple = time.localtime()                  # get struct_time
time_string = time.strftime("%m/%d/%Y, %H:%M:%S", named_tuple)

print(time_string)
```

Output

```
05/06/2020, 13:21:39
```

- Here, %Y, %m, %d, %H etc. are **format codes**:

Format Codes	Meaning [Range]
%Y	year [0001,..., 2018, 2019,..., 9999]
%m	month [01, 02, ..., 11, 12]
%d	day [01, 02, ..., 30, 31]
%H	hour [00, 01, ..., 22, 23
%M	minutes [00, 01, ..., 58, 59]
%S	second [00, 01, ..., 58, 61]

time.strptime():

- The **strptime()** function parses a string representing time and returns **struct_time**.

Example

```
import time

time_string = "6 May, 2020"
result = time.strptime(time_string, "%d %B, %Y")

print(result)
```

Output

```
time.struct_time(tm_year=2020, tm_mon=5, tm_mday=6, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=2, tm_yday=127, tm_isdst=-1)
```

math MODULE

- The **math** module is a standard module in Python and is always available.
- As its name suggests, all **mathematical operations** can be perform easily with **math** module.

Example

```
>>> import math

>>> math.sqrt(81)
9

>>> math.pi
3.141592653589793

>>> math.e
2.718281828459045

>>> math.radians(30)
0.5235987755982988

>>> math.degrees(math.pi/6)
29.999999999999996

>>> math.sin(0)
0.0

>>> math.log(10)
2.302585092994046

>>> math.exp(10)                    # gives math.e**10
1.0

>>> math.floor(8.7342)
8
```

random MODULE

- The **random** module is a standard module in Python and is always available.
- This module is used for generating **random data**.

random.random():

- Generates a random float number between **0.0 to 1.0**. The function doesn't need any arguments.

Example

```
>>> import random

>>> random.random()
0.7261258919092244
```

random.randint():

- Returns a random integer between the specified integers.

Example

```
>>> import random

>>> random.randint(1,100)
56

>>> random.randint(21,45)
43
```

random.randrange():

- Returns a randomly selected element from the range created by the **start, stop** and **step** arguments. The value of **start** is **0** by default. Similarly, the value of **step** is **1** by default.

Example

```
>>> random.randrange(1,10)
5

>>> random.randrange(1, 10, 2)
7

>>> random.randrange(1, 100, 10)
11
```

random.choice():

- Returns a **randomly selected element** from a non-empty **sequence**. An empty sequence as argument raises an **IndexError**.

Example

```
>>> random.choice('computer')
'u'

>>> random.choice([12,23,45,67,65,43])
67

>>> random.choice((12,23,45,67,65,43))
43
```

random.shuffle():

- This functions **randomly reorders the elements** in a **list**.

Example

```
>>> numbers = [12,23,45,67,65,43]

>>> random.shuffle(numbers)

>>> numbers
[23, 12, 43, 65, 67, 45]

>>> random.shuffle(numbers)

>>> numbers
[23, 43, 65, 45, 12, 67]
```

uuid MODULE

- **UUID** is a **Universally Unique Identifier**. You can also call it as GUID, i.e., Globally Unique Identifier.
- A UUID is **128 bits long number** or **ID** to uniquely identify the documents, Users, resources or information in computer systems.
- UUID can guarantee the uniqueness of Identifiers across space and time. When we talk about space and time means when UUID generated according to the standard then the identifier does not duplicate one that has already been created or will be created to identify something else.
- Therefore UUID is useful where a unique value is necessary.
- Using Python **uuid** module, you can generate versions **1, 3, 4** and **5** UUIDs. UUID generated using this module is immutable.
- Python UUID module supports the following versions of UUIDs.
    - **UUID1** – Generate UUID using a Host **MAC address**, **sequence number** and the **current time**. This version uses the IEEE 802 MAC addresses.
    - **UUID3** uses cryptographic hashing and application-provided text strings to generate UUID. UUID 3 uses **MD5** hashing.
    - **UUID4** uses **pseudo-random number generators** to generate UUID.
    - **UUID5** uses cryptographic hashing and application-provided text strings to generate UUID. UUID 5 uses **SHA-1** hashing.

Example

```
>>> import uuid

>>> u1 = uuid.uuid1()

>>> u1
UUID('2f6d1a7c-8f79-11ea-895b-820d59f08000')
```

Structure of UUID:

- As you can see in the output UUID is made up of **five components**, and each component has a fixed length. A hyphen symbol separates each component. UUID's presented in the format **"8-4-4-4-12"**.
- The formal definition of the UUID string representation is as follows:
  UUID = **time_low - time_mid - time_high_and_version - clock_seq_and_reserved_And_clock_seq_low - Node**

- This module provides immutable UUID objects (class UUID) and the functions **uuid1()**, **uuid3()**, **uuid4()**, **uuid5()** for generating version 1, 3, 4, and 5 UUIDs.

Example

```
>>> import uuid

# make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('2f6d1a7c-8f79-11ea-895b-820d59f08000')

# make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'www.rnwmultimedia.com')
UUID('84b012a9-fa34-3ba9-a998-dfb420d2d670')

# make a random UUID
>>> uuid.uuid4()
UUID('946130e4-dcff-49fd-bc63-24132cc5beea')

# make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'www.rnwmultimedia.com')
UUID('b31f84ed-9c26-5caa-a810-ed316896077f')

# make a UUID from a string of hex digits (braces and hyphens ignored)
>>> a = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> a
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')

# convert a UUID to a string of hex digits in standard form
>>> str(a)
'00010203-0405-0607-0809-0a0b0c0d0e0f'
```

NOTE:

- **uuid1 is not safe it has privacy concerns because it shows the computer's network address in UUID.**

uuid.getnode():

- To generate UUID of version 1 we need a hardware address, i.e., MAC address. It is a 48-bit positive integer.
- The **uuid.getnode()** function is used to get the MAC address of a network interface.
- If the machine has more than one network interface universally administered MAC addresses are returned instead of over locally administered MAC addresses. administered MAC addresses guaranteed to be globally unique
- if getnode() function **fails** to get MAC address it returns the **random 48-bit number** with the multicast bit as recommended in RFC 4122.

Example

```
>>> import uuid

# Get the hardware address as a 48-bit positive integer
>>> uuid.getnode()
142993855119360

# hex format
>>> hex(uuid.getnode())
'0x820d59f08000'
```

When use uuid1 and uuid4:

- **uuid1() is guaranteed not to produce any collisions**. You can create duplicates UUIDs by creating more 16384 uuid1 in less than 100ns.
- **Don't use uuid1 when you don't want to make the MAC address of your machine visible**.
- **uuid4()** uses the **cryptographically secure random number generator** to generate UUID.
- uuid4() generates a random UUID. The chance of a collision is small.
- When UUIDs require to generate on separate machines, or you want to generate a secure UUIDs use uuid4().

UUID 3 and UUID 5 to Create a Name-Based UUID:

- Version 3 or 5 UUID meant for generating UUIDs from "**names**".
- We can use name and namespace to create a series of a unique UUIDs. In simple words **version, 3 and 5 UUIDs is nothing but hashing namespace identifier with a name**.
- The **uuid.uuid3(namespace, name)** generate a UUID based on the **MD5 hash** of a namespace identifier (which is a UUID) and a string.
- Similarly, the **uuid.uuid5(namespace, name)** generate a UUID based on the **SHA-1 hashing** technique of a namespace identifier (which is a UUID) and a name.

- The UUID module defines the following namespace identifiers to use with uuid3() or uuid5().
  A. UUID.NAMESPACE_DNS means a fully qualified domain name. For example, https://pynative.com.
  B. UUID.NAMESPACE_URL When this namespace is specified, It means it is a URL.
  C. UUID.NAMESPACE_OID When this namespace is specified, the name string is an ISO OID.
  D. UUID.NAMESPACE_X500 When this namespace is specified, the name string is an X.500 DN in DER or a text output format.

Extract UUID attributes read-only attributes:

- The internal representation of a UUID is a **specific sequence of bits in memory**, as described in RFC4211. It is necessary to **convert** the **bit sequence** to a **string representation** to represent UUID in string format.
- UUID module provides the various read-only argument to access the value of each component of the UUID object.
- You can **extract** the values from UUID so we can use this value for a different purpose.

UUID Read-only Attribute:

A. **UUID.bytes**: The UUID as a 16-byte string (containing the six integer fields in big-endian byte order).
B. **UUID.bytes_le**: It is a 16-byte string that consists of a time_low, time_mid, and time_hi_version.
C. **UUID.fields**: A tuple of the six integer fields of the UUID, which are also available as six individual attributes and two derived attributes.
D. **UUID.hex**: The UUID as a 32-character hexadecimal string.
E. **UUID.int**: The integer representation of a UUID as a 128-bit integer.
F. **UUID.urn**: The UUID as a uniform resource name.
G. **UUID.variant**: The UUID variant, which determines the internal layout of the UUID. This will be one of the constants RESERVED_NCS, RFC_4122, RESERVED_MICROSOFT, or RESERVED_FUTURE.
H. **UUID.version**: the version of UUID. anything between 1, 4, 3, and 5.
I. **UUID.is_safe**: To get to know that UUID generation is safe or not.

UUID.fields:

Fields	Meaning
time_low	the first 32 bits of the UUID
time_mid	the next 16 bits of the UUID
time_hi_version	the next 16 bits of the UUID
clock_seq_hi_variant	the next 8 bits of the UUID
clock_seq_low	the next 8 bits of the UUID
node	the last 48 bits of the UUID
time	the 60-bit timestamp
clock_seq	the 14-bit sequence number

Example

```
import uuid

UUID = uuid.uuid4()

print("UUID is ", UUID)
print("UUID Type is ", type(UUID))
print('UUID.bytes   :', UUID.bytes)
print('UUID.bytes_le :', UUID.bytes_le)
print('UUID.hex     :', UUID.hex)
print('UUID.int     :', UUID.int)
print('UUID.urn     :', UUID.urn)
print('UUID.variant :', UUID.variant)
print('UUID.version :', UUID.version)
print('UUID.fields  :', UUID.fields)
print("Prining each field seperately")
print('UUID.time_low          : ', UUID.time_low)
print('UUID.time_mid          : ', UUID.time_mid)
print('UUID.time_hi_version     : ', UUID.time_hi_version)
print('UUID.clock_seq_hi_variant: ', UUID.clock_seq_hi_variant)
print('UUID.clock_seq_low       : ', UUID.clock_seq_low)
print('UUID.node              : ', UUID.node)
print('UUID.time              : ', UUID.time)
print('UUID.clock_seq         : ', UUID.clock_seq)
print('UUID.SafeUUID          : ', UUID.is_safe)
```

Output

```
UUID is  78eaedf5-d011-4a82-9097-99549df7d570
UUID Type is  <class 'uuid.UUID'>
UUID.bytes   : b'x\xea\xed\xf5\xd0\x11J\x82\x90\x97\x99T\x9d\xf7\xd5p'
UUID.bytes_le : b'\xf5\xed\xeax\x11\xd0\x82J\x90\x97\x99T\x9d\xf7\xd5p'
UUID.hex     : 78eaedf5d0114a82909799549df7d570
UUID.int     : 160727183365456721499574819069864957296
UUID.urn     : urn:uuid:78eaedf5-d011-4a82-9097-99549df7d570
UUID.variant : specified in RFC 4122
UUID.version : 4
UUID.fields  : (2028662261, 53265, 19074, 144, 151, 168588706567536)
Prining each field seperately
UUID.time_low          : 2028662261
UUID.time_mid          : 53265
UUID.time_hi_version     : 19074
UUID.clock_seq_hi_variant:  144
UUID.clock_seq_low      : 151
UUID.node              : 168588706567536
UUID.time              : 757396460813348341
UUID.clock_seq         : 4247
UUID.SafeUUID          : SafeUUID.unknown
```

#9 MODULES & PACKAGES

CREATING, IMPORTING, RENAMING & USING MODULES

- **Module** is a **file** containing a set of functions you want to include in your application.

Creating a Module:

- To create a module just save the code you want in a file with the file extension **.py**:

Example – my_module.py

```
def welcome():
    print("Welcome to Red & White")
```

Importing a Module:

- Now we can import the module we just created, by using the **import** statement:

Syntax

```
import module_name
```

- Import the module named **my_module** in any file you want like this:

Example – hello.py

```
import my_module
```

Renaming a Module:

- You can create an **alias** when you import a module, by using the **as** keyword:

Syntax

```
import module_name as alias_name
```

Example – hello.py

```
import my_module as rnw
```

Using a Module:

- Now we can use the module we just created, by using the **import** statement:
- When using a function from a module, use the syntax:

Syntax

```
module_name.function_name
```

Example – hello.py

```
import my_module
my_module.welcome()
```

Output

```
Welcome to Red & White
```

Import From Module:

- You can choose to import only parts from a module, by using the **from** keyword.

Syntax

```
from module_name import function_name
```

- The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc)
- Suppose the module named **my_module** has one function and one dictionary:

Example – my_module.py

```
def welcome():
    print("Welcome to Red & White")

person = {
    "name": "Rohan",
    "age": 23,
    "country": "India"
}
```

- Import only the **person** dictionary from the module:

Example – hello.py

```
from my_module import person

print( person['age'] )
```

Output

```
23
```

NOTE:

- When importing using the **from** keyword, do not use the module name when referring to elements in the module.
- Example: **person1["age"]**, not ~~my_module.person1["age"]~~

__name__ WITH __main__

- The **__name__** variable (two underscores before and after) is a special Python variable. It **gets its value** depending on **how we execute the containing script**.
- Sometimes you write a script with functions that might be useful in other scripts as well. In Python, you can import that script as a module in another script.
- Thanks to this special variable, **you can decide whether you want to run the script**. Or that **you want to import the functions defined in the script**.

- When you **run** your script, the **__name__** variable equals **__main__**.
- When you **import** the containing script, the **__name__** variable equals the **name of the script**.
- Let us take a look at these two use cases and describe the process with two illustrations:

Scenario 1 - Run the script:

- Suppose we wrote the script **nameScript.py** as follows:

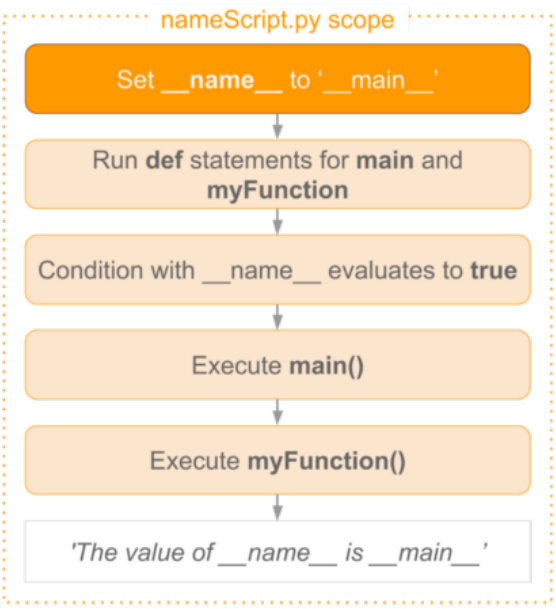
Example – nameScript.py

```
def myFunction():
    print("The value of __name__ is ", __name__)

def main():
    myFunction()

if __name__ == '__main__':
    main()
```

Output

```
The value of __name__ is __main__
```

- If you run nameScript.py, the process below is followed:



- Before all other code is run, the **__name__** variable is set to **__main__**.
- After that, the **main** and **myFunction** def statements are run. Because the condition evaluates to true, the main function is called. This, in turn, calls myFunction. This prints out the value of **__main__.**

Scenario 2 - Import the script in another script:

- If we want to re-use myFunction in another script, for example **importingScript.py**, we can import **nameScript.py** as a **module**.
- The code in **importingScript.py** could be as follows:

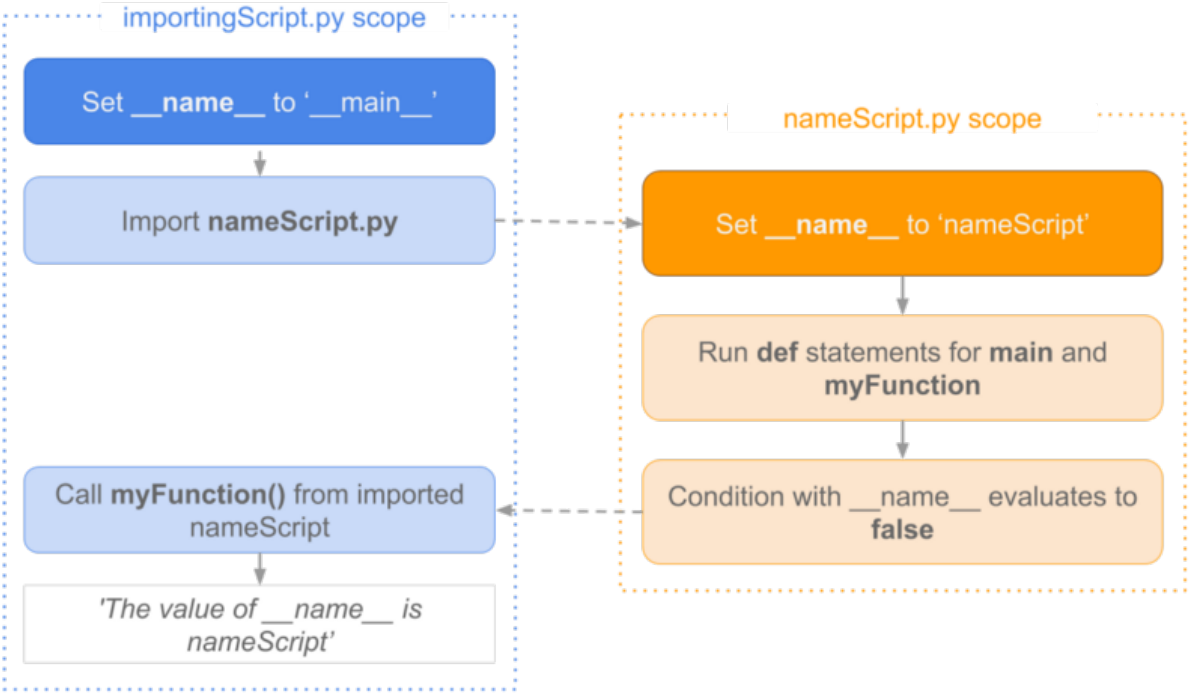
Example – importingScript.py

```
import nameScript as ns

ns.myFunction()
```

Output

```
The value of __name__ is nameScript
```

- We then have two scopes: one of **importingScript** and the second scope of **nameScript**. In the illustration, you'll see how it differs from the first use case:



- In importingScript.py the **__name__** variable is set to **__main__**.
- By importing **nameScript**, Python starts looking for a file by adding .py to the module name. It then runs the code contained in the imported file.
- But this time it is set to **nameScript**. Again the def statements for main and myFunction are run. But, now the condition evaluates to false and **main is not called**.
- In **importingScript.py** we call **myFunction** which outputs **nameScript**. NameScript is known to myFunction when that function was defined.
- If you would print **__name__** in the **importingScript**, this would output **__main__**. The reason for this is that Python uses the value known in the scope of **importingScript**.

PACKAGES & SUBPACKAGES AND dir() & __init__.py

- Suppose you have developed a very large application that includes many modules. As the number of modules grows, it becomes difficult to keep track of them all if they are dumped into one location. This is particularly so if they have similar names or functionality. You might wish for a means of grouping and organizing them.

- **Packages** allow for a hierarchical structuring of the module namespace using **dot** notation. In the same way that **modules** help avoid collisions between global variable names, **packages** help avoid collisions between module names.
- Creating a **package** is quite straightforward, since it makes use of the operating system's inherent hierarchical file structure. Consider the following arrangement:

- Here, there is a directory named **pkg** that contains two modules**, mod1.py** and **mod2.py**. The contents of the modules are:

Example – mod1.py

```
def foo():
    print('[mod1] foo()')

class Foo:
    pass
```

Example – mod2.py

```
def bar():
    print('[mod2] bar()')

class Bar:
    pass
```

- Given this structure, if the pkg directory resides in a location where it can be found, you can refer to the two modules with **dot notation** (pkg.mod1, pkg.mod2) and import them with the syntax you are already familiar with:

Syntax

```
import module_name1, …, module_nameN
```

Example

```
>>> import pkg.mod1, pkg.mod2

>>> pkg.mod1.foo()
[mod1] foo()

>>> x = pkg.mod2.Bar()
>>> x
<pkg.mod2.Bar object at 0x033F7290>
```

Syntax

```
from module_name import function_name
```

Example

```
>>> from pkg.mod1 import foo

>>> foo()
[mod1] foo()
```

Syntax

```
from module_name import function_name as alias_name
```

Example

```
>>> from pkg.mod2 import Bar as Qux

>>> x = Qux()
>>> x
<pkg.mod2.Bar object at 0x036DFFD0>
```

Syntax

```
from package_name import module_name1, …, module_nameN
from package_name import module_name as alias_name
```

Example

```
>>> from pkg import mod1

>>> mod1.foo()
[mod1] foo()

>>> from pkg import mod2 as quux

>>> quux.bar()
[mod2] bar()
```

Package Initialiaztion:

- If a file named **__init__.py** is present in a **package directory**, it is **invoked** when the **package or a module in the package is imported**. This can be used for execution of package initialization code, such as initialization of package-level data.
- For example, consider the following __init__.py file:

Example

```
print(f'Invoking __init__.py for {__name__}')
A = ['quux', 'corge', 'grault']
```

- Let's add this file to the **pkg directory** from the above example:

- Now when the package is imported, the global list A is initialized:

Example

```
>>> import pkg
Invoking __init__.py for pkg

>>> pkg.A
['quux', 'corge', 'grault']
```

- A **module** in the package can access the global variable by importing it in turn:

Example – mod1.py

```
def foo():
    from pkg import A
    print('[mod1] foo()')

class Foo:
    pass
```

Example

```
>>> from pkg import mod1
Invoking __init__.py for pkg

>>> mod1.foo()
[mod1] foo() / A =  ['quux', 'corge', 'grault']
```

- **__init__.py can also be used to effect automatic importing of modules from a package.**
- For example, earlier you saw that the statement import **pkg** only places the name **pkg** in the caller's local symbol table and doesn't import any modules.
- But if **__init__.py** in the **pkg** directory contains the following:

Example - __init__.py

```
print(f'Invoking __init__.py for {__name__}')
import pkg.mod1, pkg.mod2
```

- then when you execute import **pkg**, modules **mod1** and **mod2** are imported automatically:

Example

```
>>> import pkg
Invoking __init__.py for pkg

>>> pkg.mod1.foo()
[mod1] foo()

>>> pkg.mod2.bar()
[mod2] bar()
```

NOTE:

- Much of the Python documentation states that an **__init__.py** file **must** be present in the package directory when creating a package. This was once true. It used to be that the very presence of **__init__.py** signified to Python that a package was being defined. The file could contain initialization code or even be empty, but it had to be present.

- Starting with **Python 3.3**, **Implicit Namespace Packages** were introduced. These allow for the creation of a package **without any __init__.py** file. Of course, it **can** still be present if package initialization is needed. But it is no longer required.

Importing * From a Package:

- For the purposes of the following discussion, the previously defined package is expanded to contain some additional modules:

- There are now four modules defined in the **pkg** directory. Their contents are as shown below:

Example – mod1.py

```
def foo():
    print('[mod1] foo()')

class Foo:
    pass
```

Example – mod2.py

```
def bar():
    print('[mod2] bar()')

class Bar:
    pass
```

Example – mod3.py

```
def baz():
    print('[mod3] baz()')

class Baz:
    pass
```

Example – mod4.py

```
def qux():
    print('[mod4] qux()')

class Qux:
    pass
```

- You have already seen that when **import *** is used for a **module**, all objects from the module are imported into the local symbol table, except those whose names begin with an underscore, as always:

Example

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__',
'__package__', '__spec__']

>>> from pkg.mod3 import *

>>> dir()
['Baz', '__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__',
'__package__', '__spec__', 'baz']

>>> baz()
[mod3] baz()

>>> Baz
<class 'pkg.mod3.Baz'>
```

- There The analogous statement for a package is this:

Syntax

```
from package_name import *
```

- What does that do?

Example

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__',
'__package__', '__spec__']

>>> from pkg import *

>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__',
'__package__', '__spec__']
```

- Hmph. Not much. You might have expected (assuming you had any expectations at all) that Python would dive down into the package directory, find all the modules it could, and import them all. But as you can see, by default that is not what happens.
- Instead, Python follows this convention: if the **__init__.py** file in the **package** directory contains a **list** named **__all__**, it is taken to be a list of modules that should be imported when the statement **from <package_name> import *** is encountered.
- For the present example, suppose you create an __init__.py in the pkg directory like this:

Example - __init__.py

```
__all__ = [
      'mod1',
      'mod2',
      'mod3',
      'mod4'
      ]
```

- Now **from pkg import * imports all four modules**:

Example

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__']

>>> from pkg import *

>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', 'mod1', 'mod2', 'mod3',
'mod4']

>>> mod2.bar()
[mod2] bar()

>>> mod4.Qux
<class 'pkg.mod4.Qux'>
```

- Using **import *** still isn't considered terrific form, any more for packages than for modules.
- But this facility at least gives the creator of the package some control over what happens when import * is specified. (In fact, it provides the capability to disallow it entirely, simply by declining to define __all__ at all. As you have seen, the default behaviour for packages is to import nothing.)
- By the way, **__all__** can be defined in a **module** as well and serves the same purpose: to control what is imported with import *.
- For example, modify **mod1.py** as follows:

Example – mod1.py

```
__all__ = [ 'foo' ]

def foo():
    print('[mod1] foo()')

class Foo:
    pass
```

- Now an **import *** statement from **pkg.mod1** will only import what is contained in **__all__**:

Example

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__']

>>> from pkg.mod1 import *

>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__', 'foo']

>>> foo()
[mod1] foo()

>>> Foo
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    Foo
NameError: name 'Foo' is not defined
```

- foo() (the function) is now defined in the local namespace, but **Foo** (the class) is not, because the latter is not in **__all__**.
- In summary, __all__ is used by both **packages** and **modules** to control what is imported when import * is specified. But the default behavior differs:
  A. **For a package**, when __all__ is not defined, **import * does not import anything**.
  B. **For a module**, when __all__ is not defined, **import * imports everything** (except-you guessed it-names starting with an underscore).

Subpackages:

- Packages can contain nested **subpackages** to arbitrary depth. For example, let's make one more modification to the example **package** directory as follows:

- The four modules (mod1.py, mod2.py, mod3.py and mod4.py) are defined as previously. But now, instead of being lumped together into the **pkg directory**, they are split out into two **subpackage** directories, **sub_pkg1** and **sub_pkg2**.
- Importing still works the same as shown previously. Syntax is similar, but additional **dot notation** is used to separate **package** name from **subpackage** name:

Example

```
>>> import pkg.sub_pkg1.mod1
>>> pkg.sub_pkg1.mod1.foo()
[mod1] foo()

>>> from pkg.sub_pkg1 import mod2
>>> mod2.bar()
[mod2] bar()

>>> from pkg.sub_pkg2.mod3 import baz
>>> baz()
[mod3] baz()

>>> from pkg.sub_pkg2.mod4 import qux as grault
>>> grault()
[mod4] qux()
```

- In addition, a module in one **subpackage** can reference objects in a **sibling subpackage** (in the event that the sibling contains some functionality that you need).
- For example, suppose you want to import and execute function foo() (defined in module mod1) from within module mod3. You can either use an **absolute import**:

Example - mod3.py

```
def baz():
    print('[mod3] baz()')

class Baz:
    pass

from pkg.sub_pkg1.mod1 import foo
foo()
```

Example

```
>>> from pkg.sub_pkg2 import mod3
[mod1] foo()

>>> mod3.foo()
[mod1] foo()
```

- Or you can use a **relative import**, where **..** refers to the package one level up. From within mod3.py, which is in subpackage sub_pkg2,
  A. **..** evaluates to the parent package (pkg), and
  B. **..sub_pkg1** evaluates to subpackage sub_pkg1 of the parent package.

Example - mod3.py

```
def baz():
    print('[mod3] baz()')

class Baz:
    pass

from .. import sub_pkg1
print(sub_pkg1)

from ..sub_pkg1.mod1 import foo
foo()
```

Example

```
>>> from pkg.sub_pkg2 import mod3
<module 'pkg.sub_pkg1' (namespace)>
[mod1] foo()
```

#10 REGULAR EXPRESSION (RegEx) & CLA

re MODULE

- A **RegEx**, or **Regular Expression**, is a **sequence of characters that forms a search pattern**.
- RegEx can be used to check if a string contains the specified search pattern.
- For instance, a regular expression could tell a program to search for specific text from the string and then to print out the result accordingly. Expression can include
    - Text matching
    - Repetition
    - Branching
    - Pattern-composition etc.
- In Python, a regular expression is denoted as RE (REs, regexes or regex pattern) are imported through **re module**. Python supports regular expression through libraries.
- In Python regular expression supports various things like **Metacharacters**, **Special Sequences**, and **Sets**.

Metacharacters:

- Metacharacters are characters with a special meaning:

Character	Description	Example
[ ]	A set of characters	"[a-m]"
\	Signals a special sequence (can also be used to escape special characters)	"\d"
.	Any character (except newline character)	"he..o"
^	Starts with	"^hello"
$	Ends with	"world$"
*	Zero or more occurrences	"aix*"
+	One or more occurrences	"aix+"
{ }	Exactly the specified number of occurrences	"al{2}"
|	Either or	"falls|stays"
( )	Capture and group	-

Special Sequences:

- A special sequence is a **** followed by one of the characters in the list below, and has a special meaning:

Character	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"
\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"

Sets:

- A set is a set of characters inside a pair of square brackets **[ ]** with a special meaning:

Set	Description
[arn]	Returns a match where one of the specified characters (a, r, or n) are present
[a-n]	Returns a match for any lower case character, alphabetically between a and n
[^arn]	Returns a match for any character EXCEPT a, r, and n
[0123]	Returns a match where any of the specified digits (0, 1, 2, or 3) are present
[0-9]	Returns a match for any digit between 0 and 9
[0-5][0-9]	Returns a match for any two-digit numbers from 00 and 59
[a-zA-Z]	Returns a match for any character alphabetically between a and z, lower case OR upper case
[+]	In sets, +, *, ., |, (), $,{} has no special meaning, so [+] means: return a match for any + character in the string

- The **re module** offers a set of **functions** that allows us to search a string for a match:
  - search()
  - findall()
  - split()
  - sub()

A. search()

- The **search()** function searches the string for a match, and returns a **Match object** if there is a match.
- If there is more than one match, **only the first occurrence of the match will be returned**:

Example

```
import re

txt = "Red & White Group of Institutes"
x = re.search("\s", txt)

print(“Result: ”, x)
print("The first white-space character is located in position:", x.start())
```

Output

```
Result:  <re.Match object; span=(3, 4), match=' '>
The first white-space character is located in position: 3
```

- If no matches are found, the value **None** is returned:

Example

```
import re

txt = "Red & White Group of Institutes"
x = re.search("Multimedia", txt)

print(“Result: ”, x)
```

Output

```
Result:  None
```

B. findall()

- The **findall()** function returns a **list containing all matches**.

Example

```
import re

txt = "Red & White Group of Institutes"
x = re.findall("it", txt)

print(“Result: ”, x)
```

Output

```
Result:  <re.Match object; span=(8, 10), match='it'>
```

- If no matches are found, an **empty list** is returned:

Example

```
import re

txt = "Red & White Group of Institutes"
x = re.findall("Multimedia", txt)

print(“Result: ”, x)
```

Output

```
Result: [ ]
```

C. split()

- The **split()** function returns a **list** where the **string has been split at each match**:

Example

```
import re

txt = "Red & White Group of Institutes"
x = re.split("\s", txt)

print(“Result: ”, x)
```

Output

```
Result: [‘Red’, ‘&’, ‘White’, ‘Group’, ‘of’, ‘Institute’]
```

- You can control the number of occurrences by specifying the **maxsplit** parameter:

Example

```
import re

txt = "Red & White Group of Institutes"
x = re.split("\s", txt, maxsplit=2)

print(“Result: ”, x)
```

Output

Result: **['Red', '&', 'White Group of Institute']**

D. sub()

- The **sub()** function **replaces** the **matches with the text of your choice**:

Example

```
import re

txt = "Red & White Group of Institutes"
x = re.sub("\s", "-", txt)

print("Result: ", x)
```

Output

Result: **Red-&-White-Group-of-Institute**

- You can control the number of replacements by specifying the **count** parameter:

Example

```
import re

txt = "Red & White Group of Institutes"
x = re.sub("\s", "-", txt, count=2)

print("Result: ", x)
```

Output

Result: **Red-&-White Group of Institute**

Match Object:

- A Match Object is an object containing **information about the search and the result**.
- If there is no match, the value **None** will be returned, instead of the Match Object.
- The Match object has properties and methods used to retrieve information about the search, and the result:
  A. **.span()** returns a tuple containing the start-, and end positions of the match.
  B. **.string** returns the string passed into the function
  C. **.group()** returns the part of the string where there was a match

Example

```
import re

txt = "Red & White Group of Institutes"

# RegEx looks for any words that starts with an upper case "G":
x = re.search(r"\bG\w+", txt)

print(“Result: ”, x.span())
```

Output

```
Result:  (12, 17)
```

Example

```
import re

txt = "Red & White Group of Institutes"

# RegEx looks for any words that starts with an upper case "G":
x = re.search(r"\bG\w+", txt)

print(“Result: ”, x.string)
```

Output

```
Result:  Red & White Group of Institutes
```

Example

```
import re

txt = "Red & White Group of Institutes"

# RegEx looks for any words that starts with an upper case "W":
x = re.search(r"\bW\w+", txt)

print("Result: ", x.group())
```

Output

```
Result:  White
```

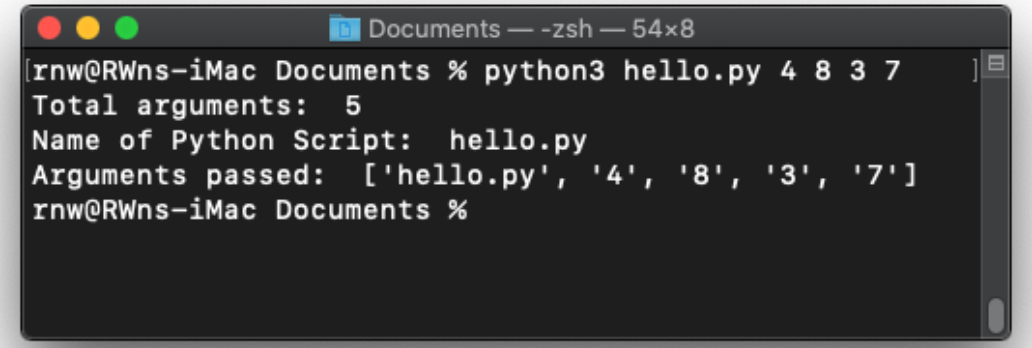
COMMAND LINE ARGUMENTS (CLA)

- The **arguments** that are given after the name of the program in the **command line shell** of the operating system are known as **Command Line Arguments**.
- The **sys** module provides functions and variables used to manipulate different parts of the Python runtime environment. This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.
- One such variable is **sys.argv** which is a simple list structure. It's main purpose are:
    - It is a **list** of **command line arguments**.
    - **len(sys.argv)** provides the number of command line arguments.
    - **sys.argv[0]** is the name of the **current Python script**.

Example

```
import sys

n = len(sys.argv)
print("Total arguments: ", n)

print("Name of Python Script: ", sys.argv[0])

print("Arguments passed: ", sys.argv)
```

Output


Documents — -zsh — 54×8
rnw@RWns-iMac Documents % python3 hello.py 4 8 3 7
Total arguments: 5
Name of Python Script: hello.py
Arguments passed: ['hello.py', '4', '8', '3', '7']
rnw@RWns-iMac Documents %

#11 PIP - PACKAGE MANAGER & DATABASE INTERACTION

WHAT IS SQL?

- **SQL** is a standard language for storing, manipulating and retrieving data in **databases**.
- SQL stands for **Structured Query Language**
- SQL lets you access and manipulate databases
- **SQL can execute queries against a database**
- **SQL can retrieve data from a database**
- **SQL can insert records in a database**
- **SQL can update records in a database**
- **SQL can delete records from a database**
- **SQL can create new databases**
- **SQL can create new tables in a database**
- **SQL can create stored procedures in a database**
- **SQL can create views in a database**
- **SQL can set permissions on tables, procedures, and views**

RDBMS:

- **RDBMS** stands for **Relational Database Management System**.
- RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.
- The data in RDBMS is stored in database objects called **tables**.
- **A table is a collection of related data entries and it consists of columns and rows**.
- Every table is broken up into smaller entities called **fields**.
- A **field** is a **column** in a table that is designed to maintain specific information about every record in the table.
- A **record**, also called a **row**, is each individual entry that exists in a table. A record is a horizontal entity in a table.
- A column is a vertical entity in a table that contains all information associated with a specific field in a table.

SQL QUERIES

- **SQL queries** are useful for **retrieving, manipulating** or **deleting** data from tables. Using SQL queries, you can also manipulate table itself.
- Some database systems require a **semicolon** at the **end of each SQL statement**.
- Semicolon is the standard way to separate each SQL statement in database systems that allow more than one SQL statement to be executed in the same call to the server.
- Some of the most important SQL commands:
    - **SELECT** - extracts data from a database
    - **UPDATE** - updates data in a database
    - **DELETE** - deletes data from a database
    - **INSERT INTO** - inserts new data into a database
    - **CREATE DATABASE** - creates a new database
    - **DROP DATABASE** - deletes a database
    - **CREATE TABLE** - creates a new table
    - **ALTER TABLE** - modifies a table
    - **DROP TABLE** - deletes a table

Syntax

```
SELECT * FROM table_name;
```

```
CREATE DATABASE database_name;
```

```
CREATE TABLE table_name(
        column1 datatype,
        column2 datatype,
        column3 datatype,
);
```

XAMPP

- **XAMPP** is a free and open-source cross-platform web server solution stack package developed by **Apache Friends**.
- The abbreviation or acronym of XAMPP is stands for:
  - **X - Cross-platform**
  - **A - Apache Server**
  - **M - MariaDB /MySQL**
  - **P - PHP**
  - **P - Perl**

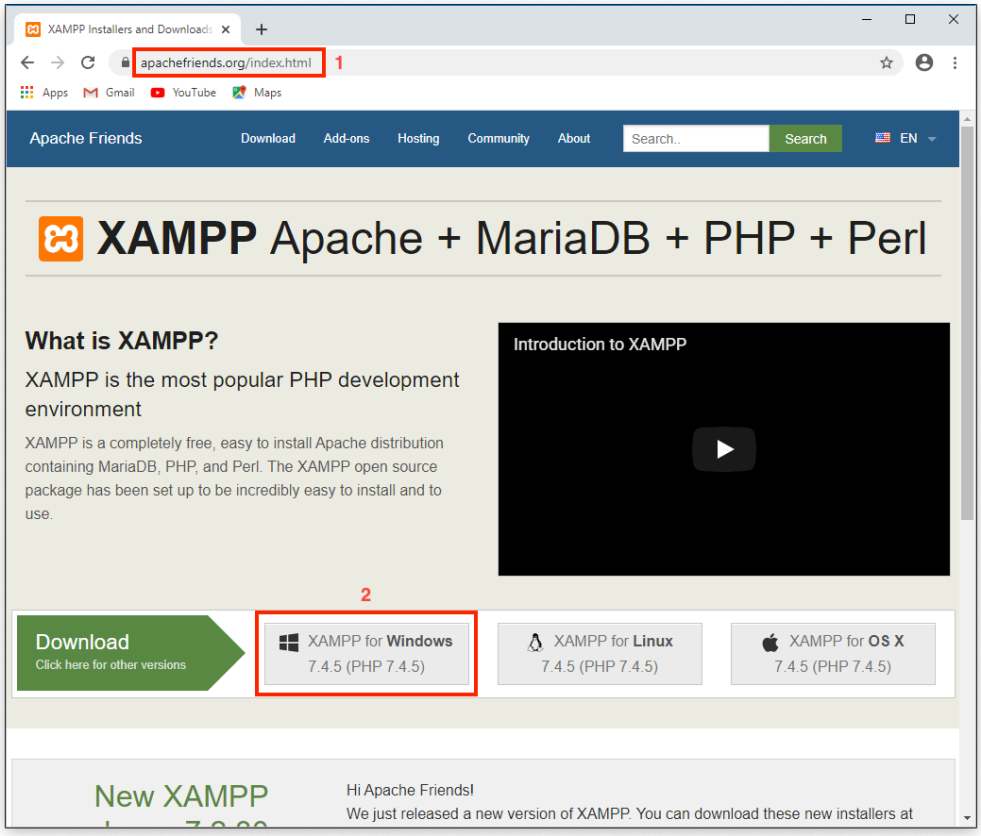
- We use MySQL database for database interaction with Python.
- But first, we perform all SQL operations on **phpMyAdmin** interface which is come built-in with **XAMPP**.

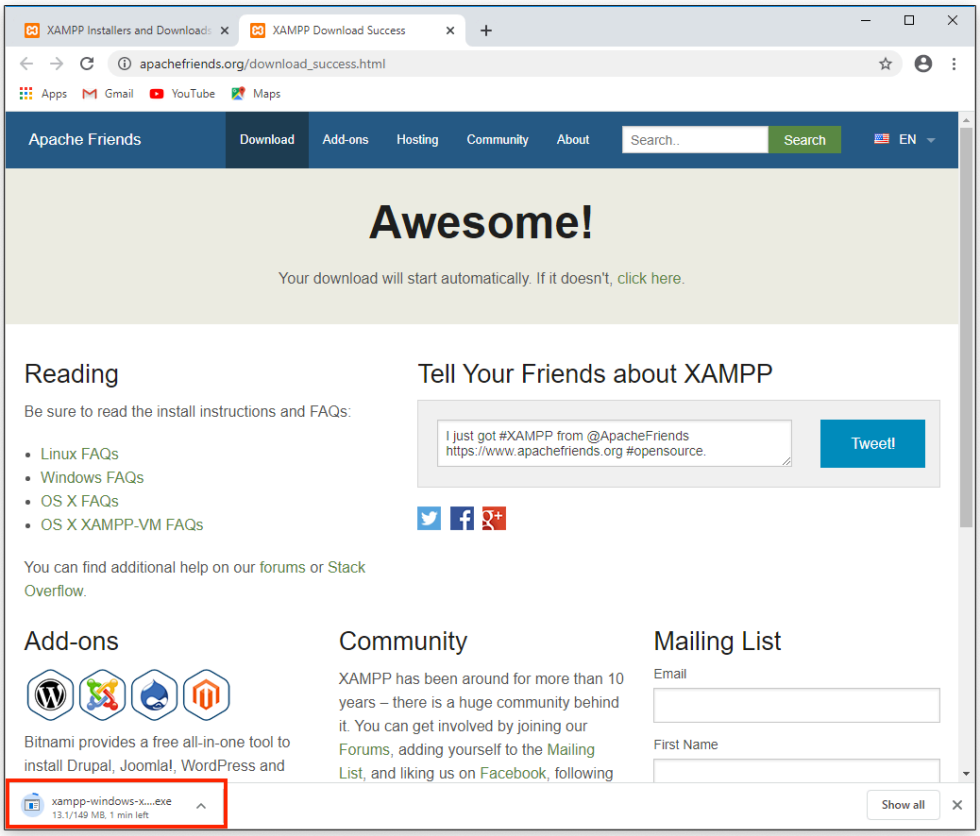
INSTALLATION OF XAMPP:

STEP - 1 First go to this link: https://www.apachefriends.org/index.html

STEP - 2

Click on **XAMPP for Windows**



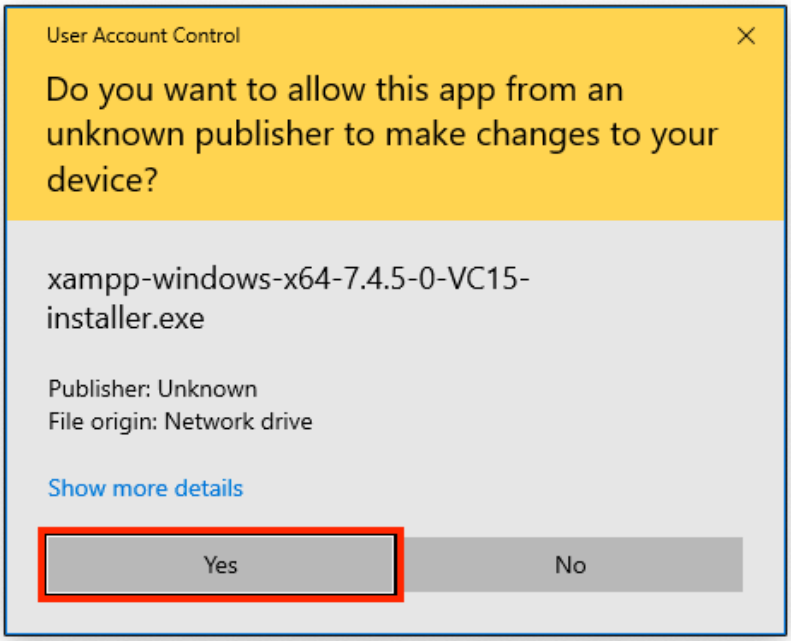


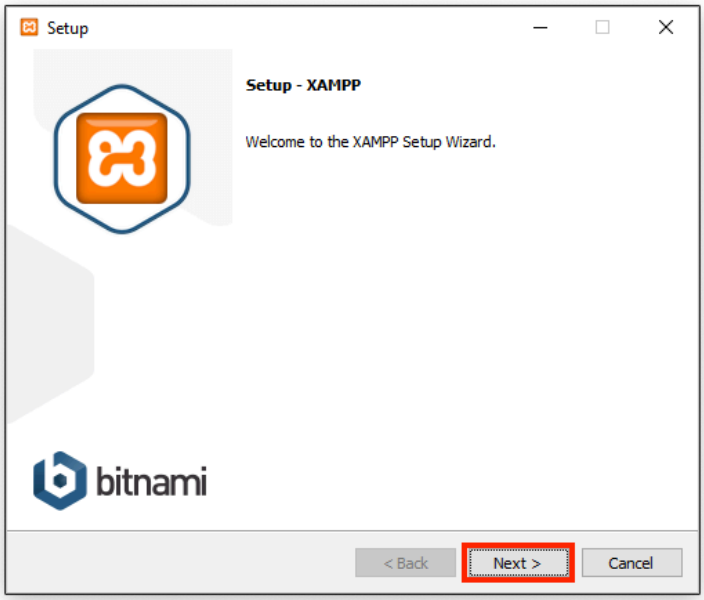
STEP - 3

Download will start.

STEP - 4

Open downloaded file.
Click on **Yes** button.



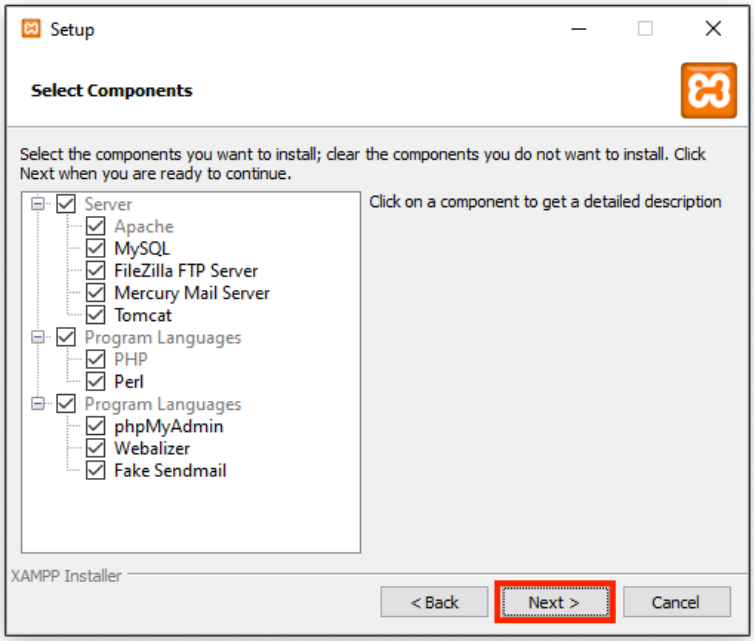


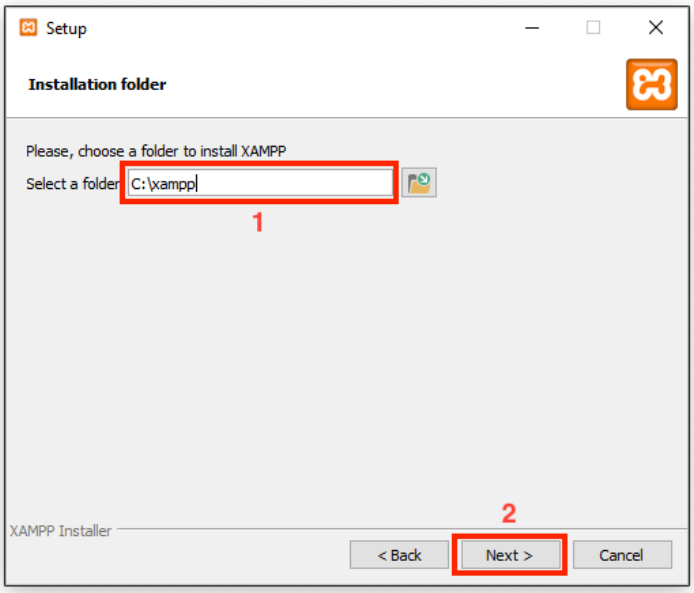
STEP - 5

Click on **Next >** button.

STEP - 6

Click on **Next >** button.



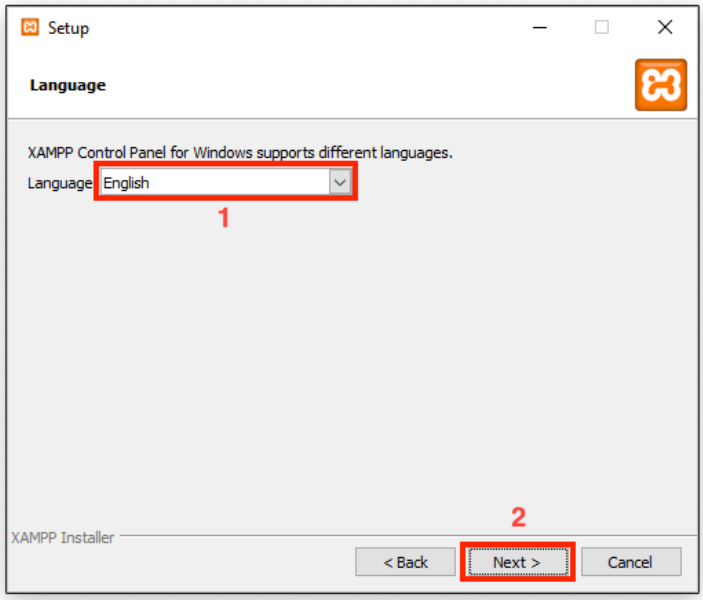


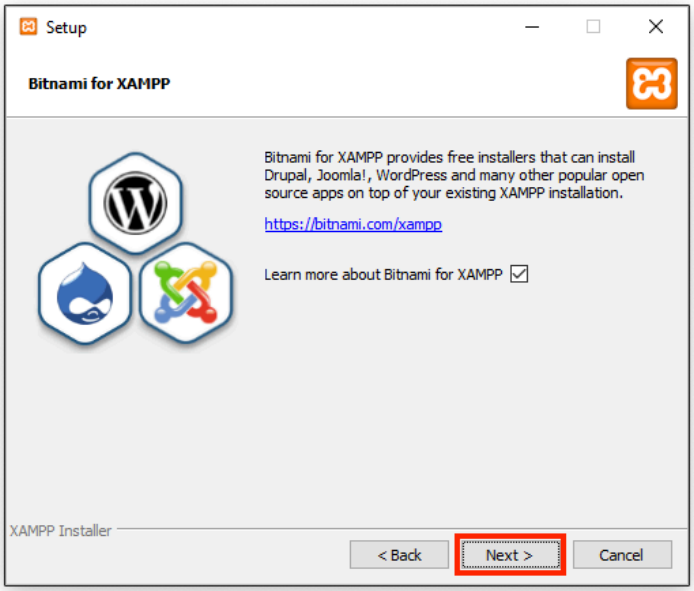
STEP - 7

Select your installation path.
Then click on **Next >** button.

STEP - 8

Choose your language for XAMPP interface.
Then, click on **Next >** button.



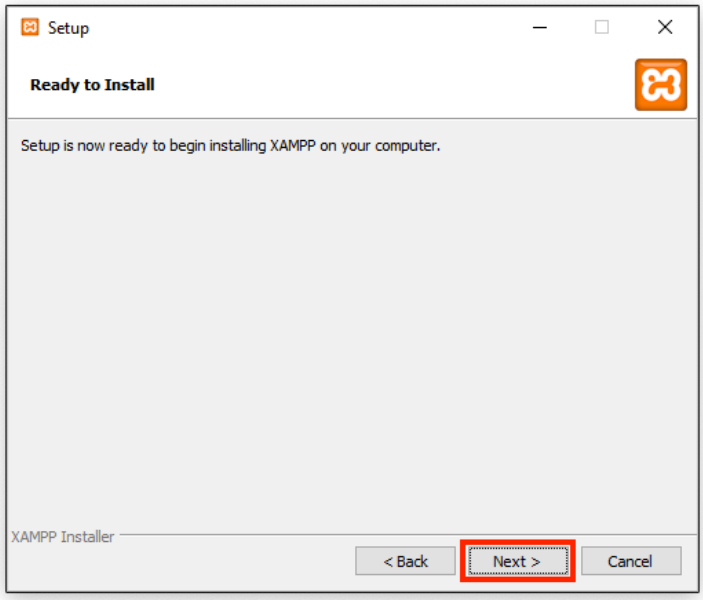


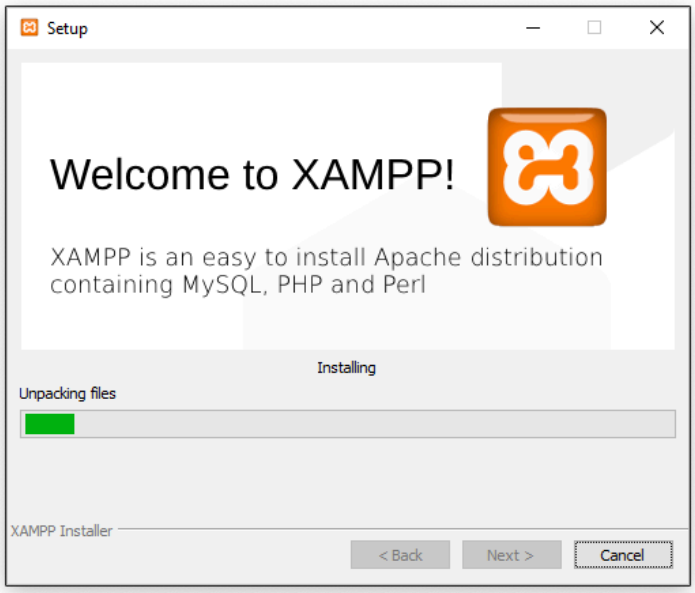
STEP - 9

Click on **Next >** button.

STEP - 10

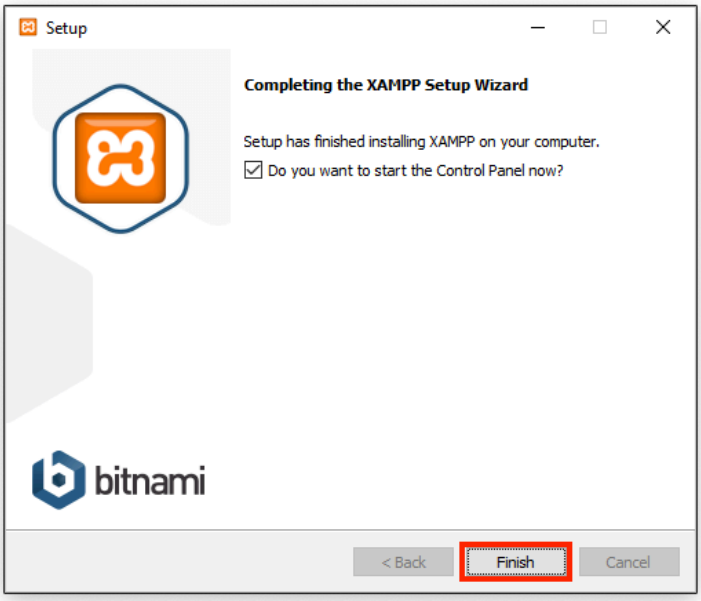
Click on **Next >** button.





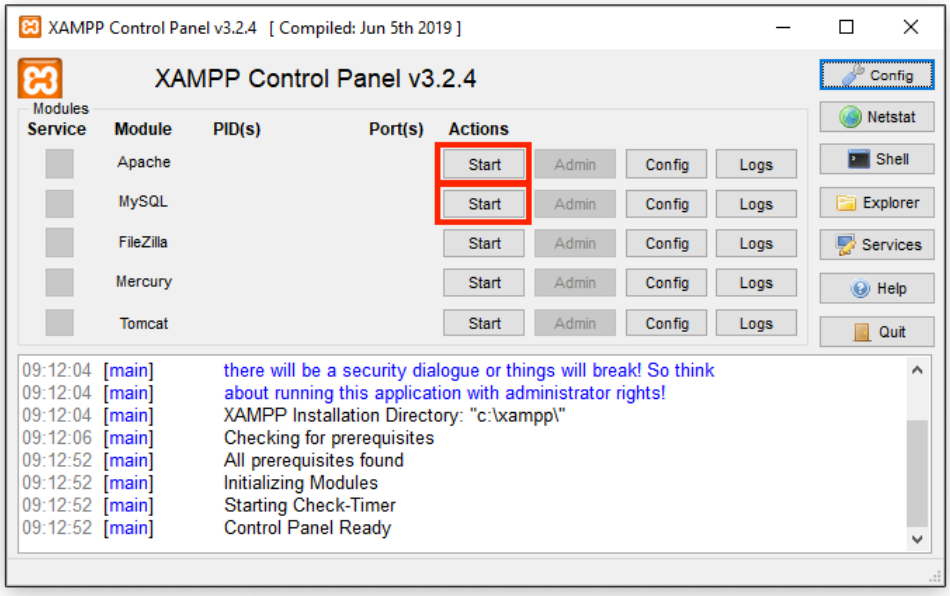
STEP - 11

Setup is now installing.



STEP - 12

Click on **Finish** button.

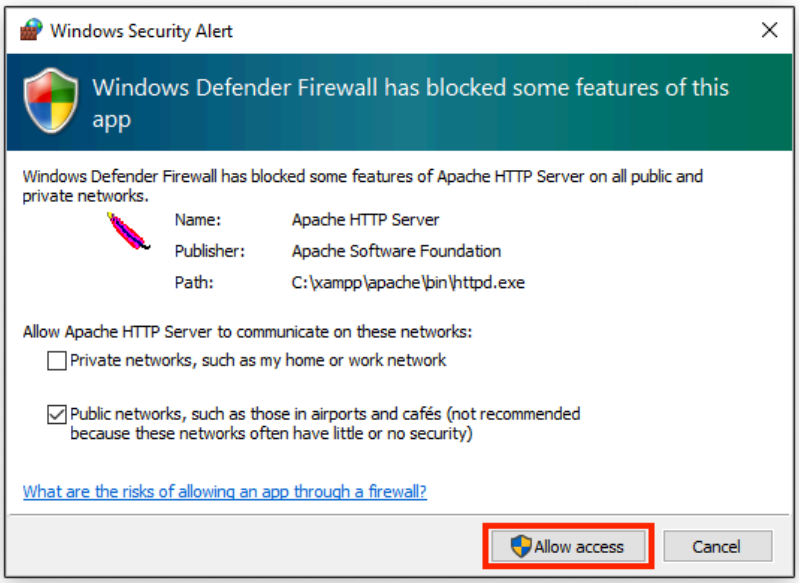


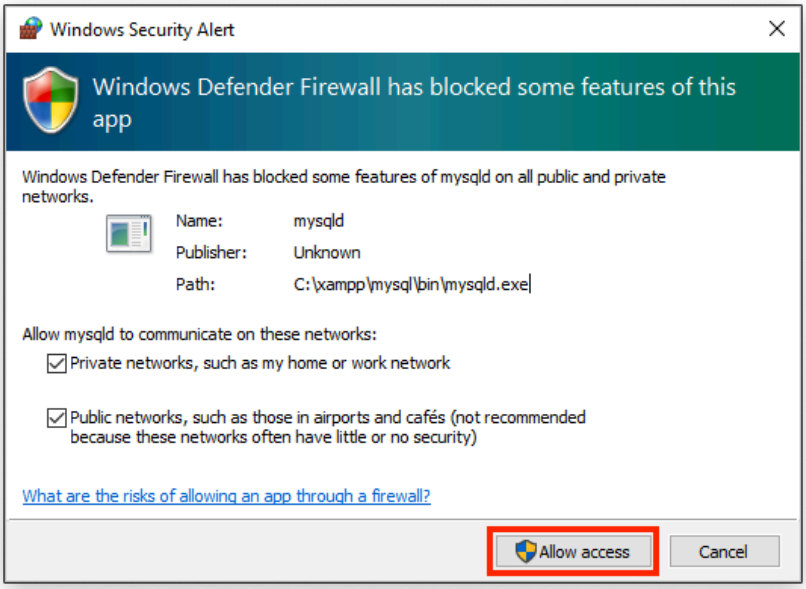
STEP - 13

Click on **Start** button of Apache and MySQL.

STEP - 14

Click on **Allow access** button.





STEP - 15

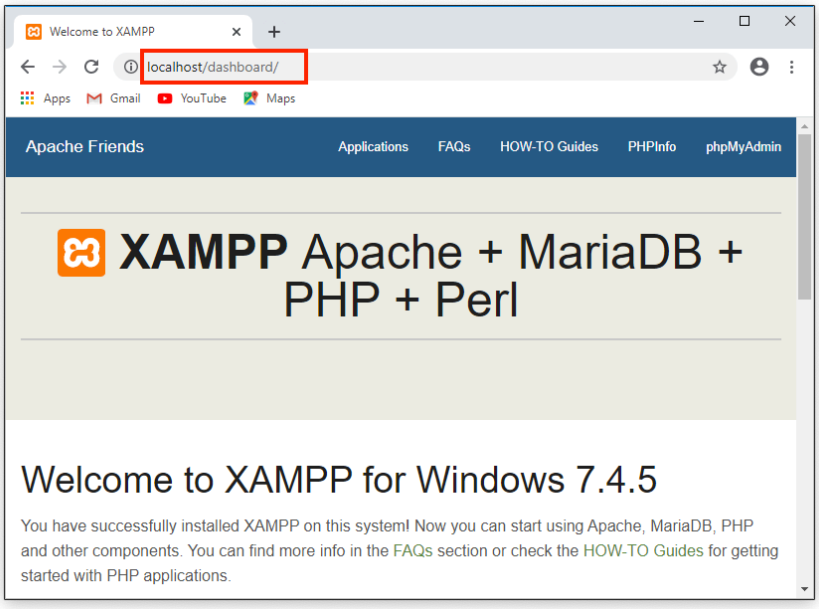
Click on **Allow access** button.

STEP - 16

Now open your web browser such as Chrome.

Enter **localhost** in address bar and hit enter.

You can also enter IP address of localhost that is **127.0.0.1**



CRUD OPERATION WITH XAMPP (phpMyAdmin)

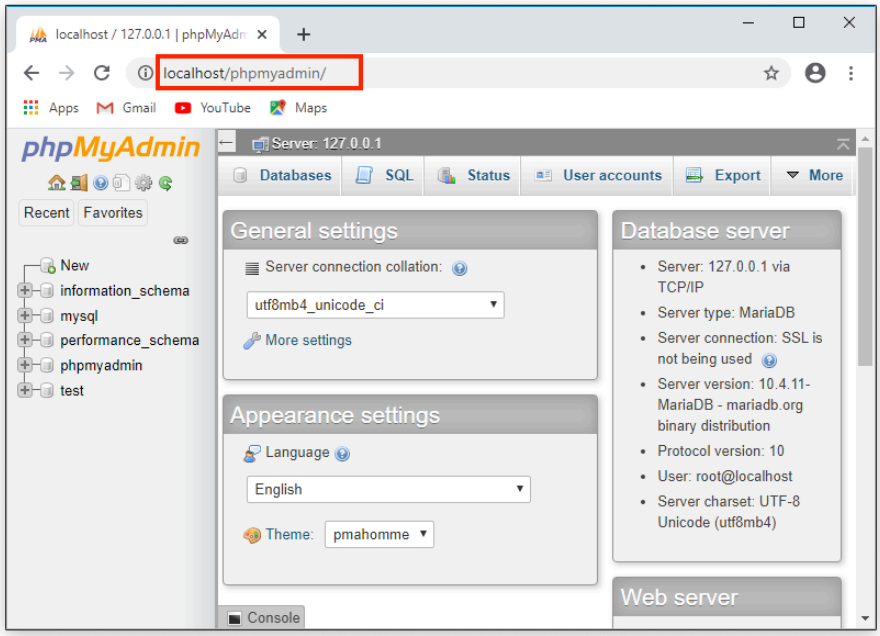
- **phpMyAdmin** is a tool provided with **XAMPP** to perform database related operations.
- First we perform all database related operations in phpMyAdmin interface.
- The abbreviation or acronym of CRUD is stands for:
  - **C - Create**
  - **R - Retrieve**
  - **U - Update**
  - **D - Delete**

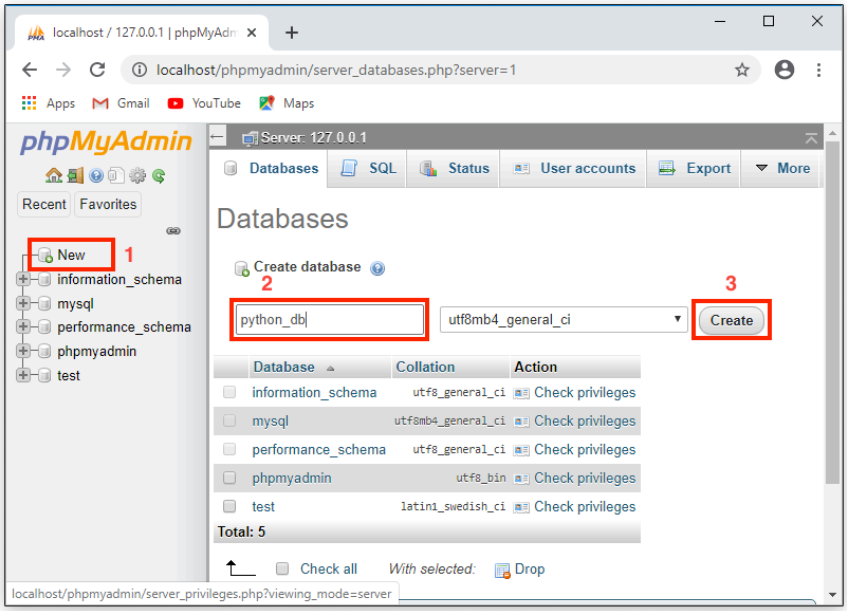
- First let's open phpMyAdmin:
  -

STEP - 1

Now open your web browser such as Chrome.

Enter **localhost/phpmyadmin** in address bar and hit enter.





STEP - 2

Click on **New** button.

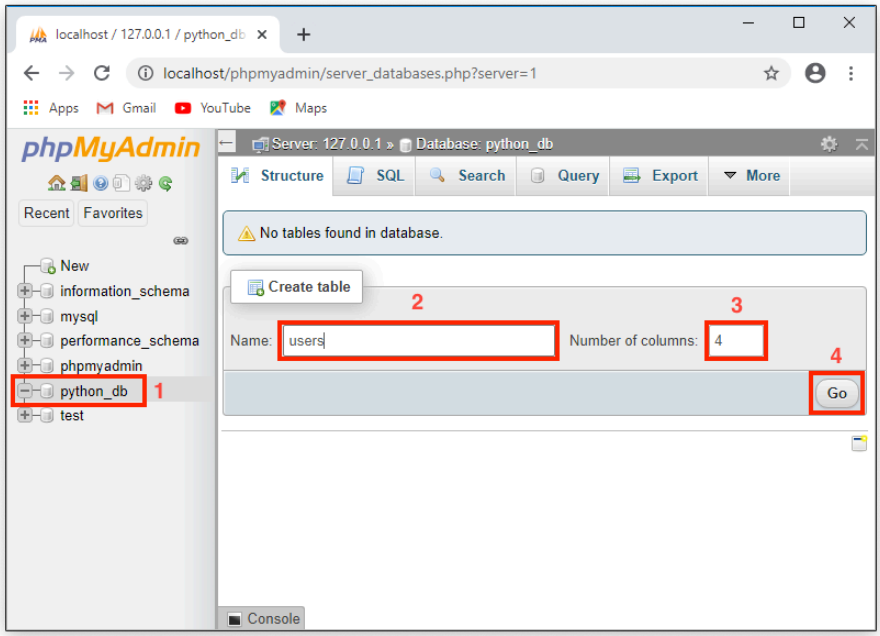
Enter **database_name** and click on **Create** button.

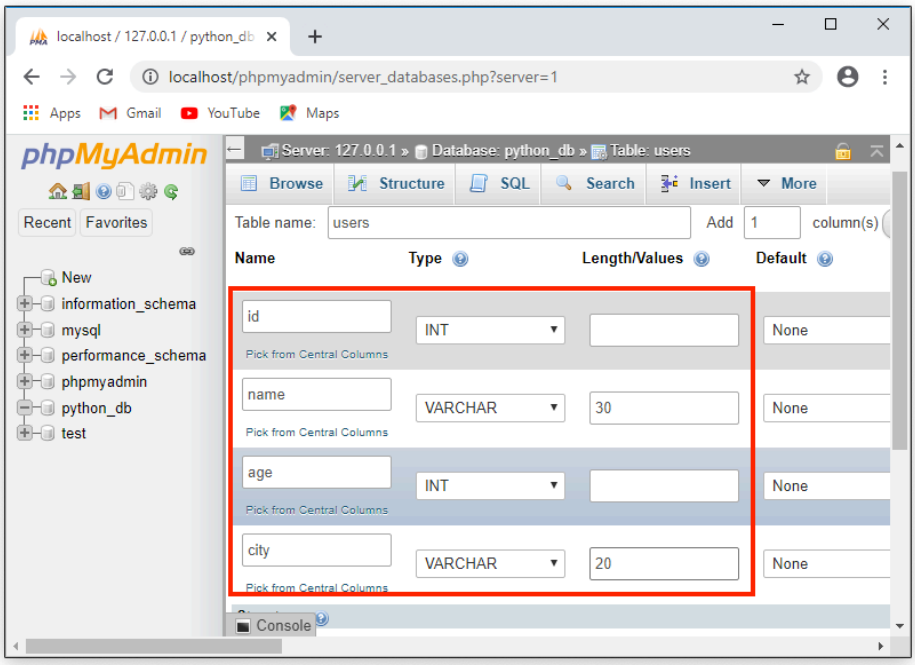
STEP - 3

Our created database shows up on left sidebar.

Now enter new table name and no. columns you want.

Then, click on **Go** button to create a table.



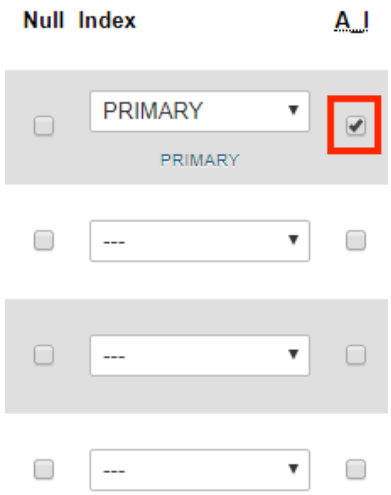


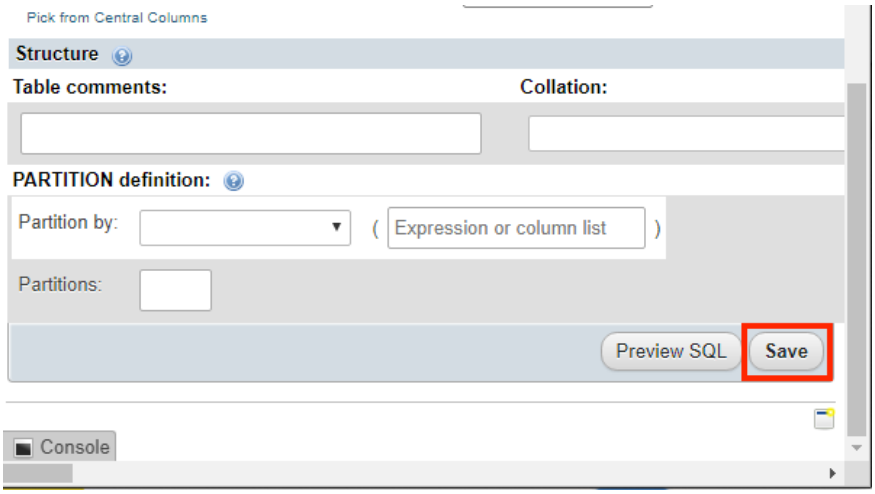
STEP - 4

Enter column name and type and also length as shown in picture.

STEP - 5

Click on **A_I** checkbox of **id** column for applying unique and auto_increment value.





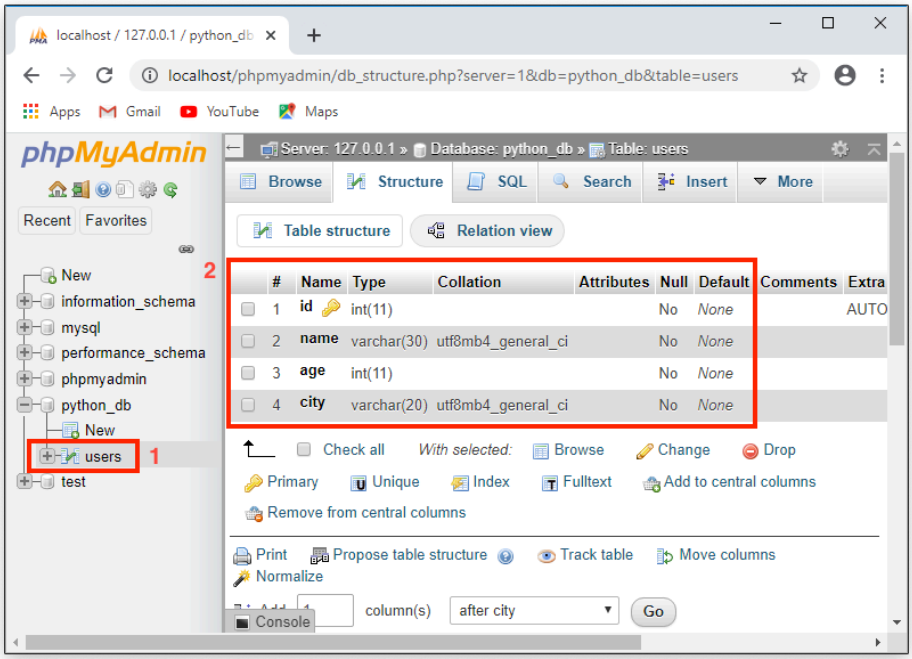
STEP - 6

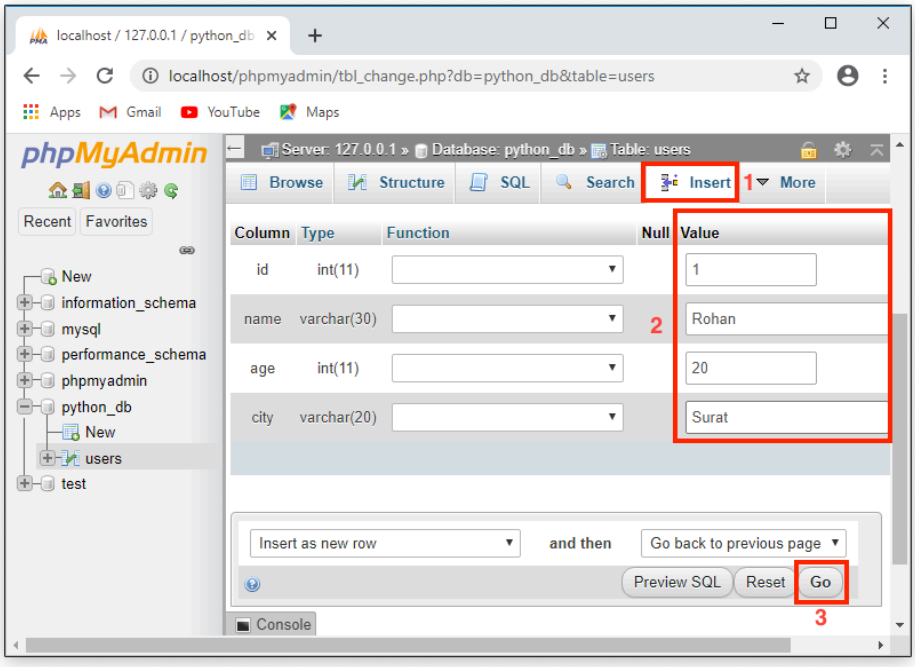
Now, click on **Save** button.

STEP - 7

Our created table shows up on left sidebar.

Now you can show all table structure.





STEP - 8

No, we insert some records by repeating this process:

Go to **insert** section.

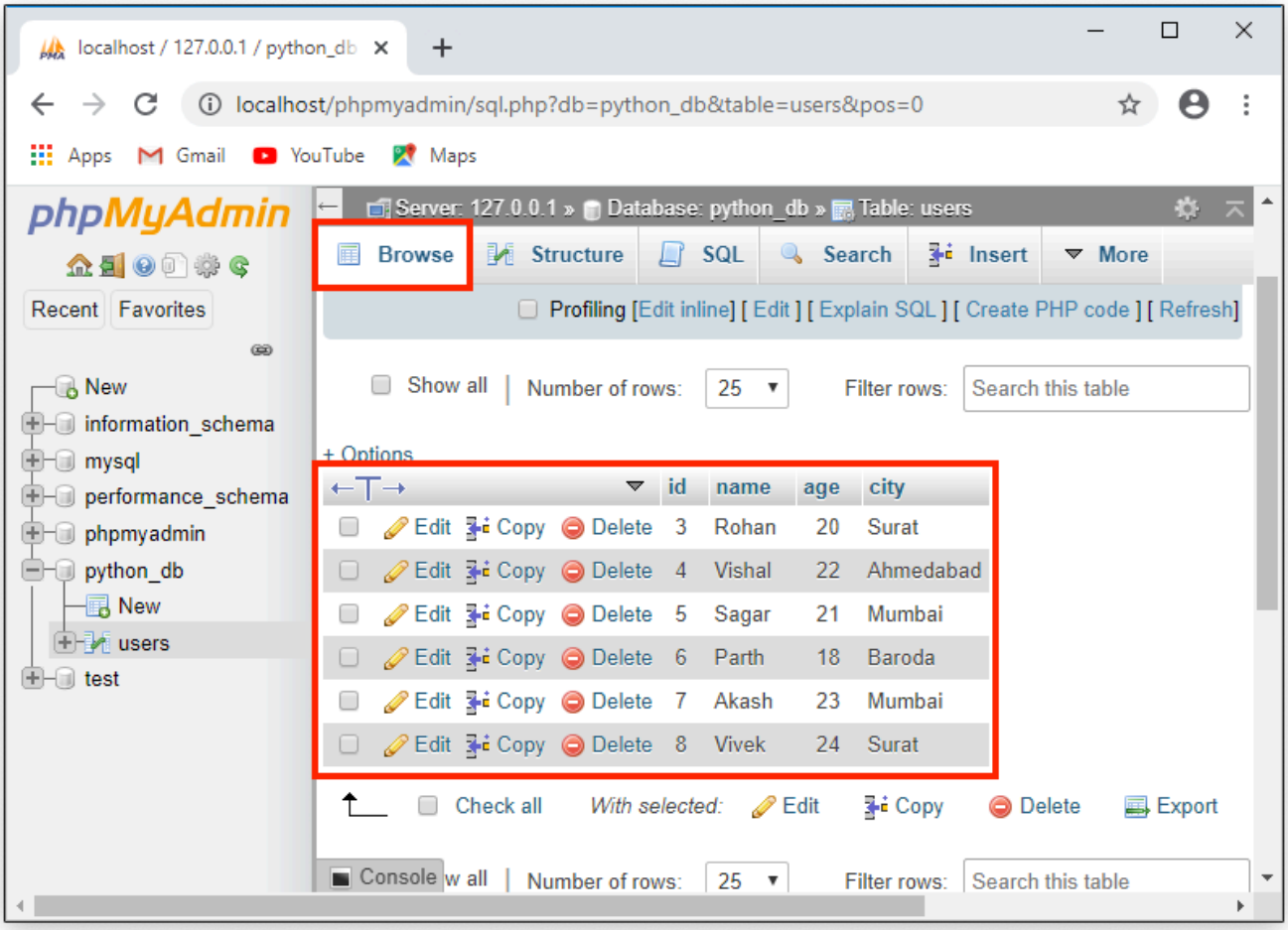
Enter all values.

Click on **Go** button.

STEP - 9

After inserting some records, click on **Browse** section to view all your records.

In Browse section, you can **edit** and **delete** each record you want.



NOTE:

- **All above process can be done using only SQL queries also. We will discuss this in future sections.**

PIP - PYTHON PACKAGE MANAGER

- **PIP** is a **package manager** for Python **packages**, or **modules** if you like.
- A package contains all the files you need for a module.
- Modules are Python code libraries you can include in your project.

NOTE:

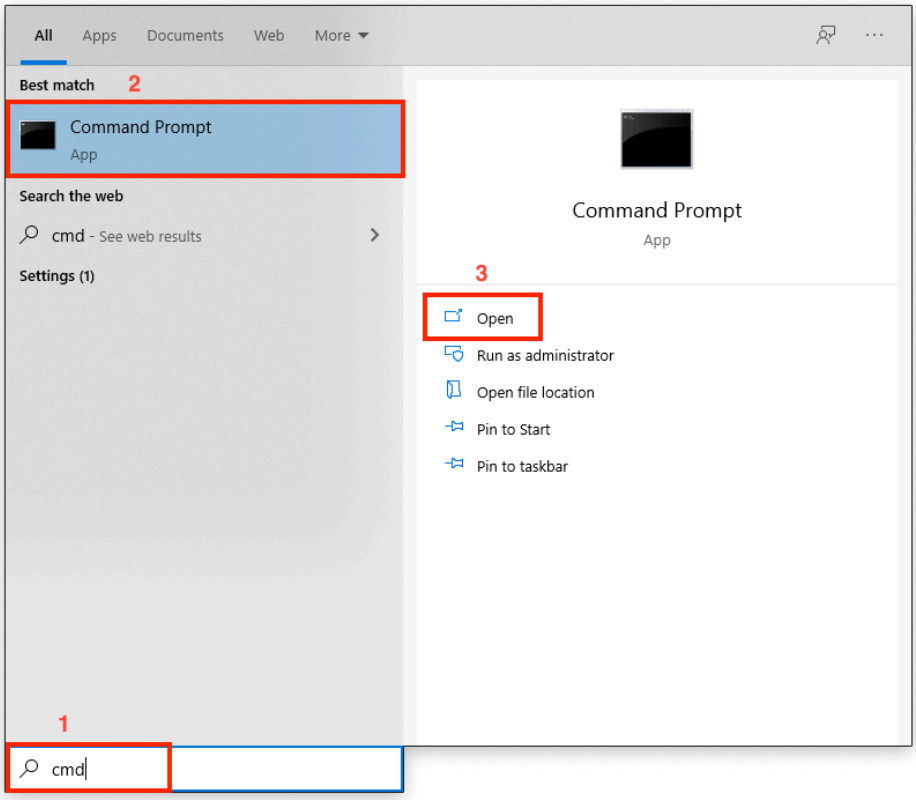
- **PIP is included by default in Python version 3.4 or later.**

Check if PIP is installed:

STEP - 1

Search **cmd** in start menu.

Click on **Open**.



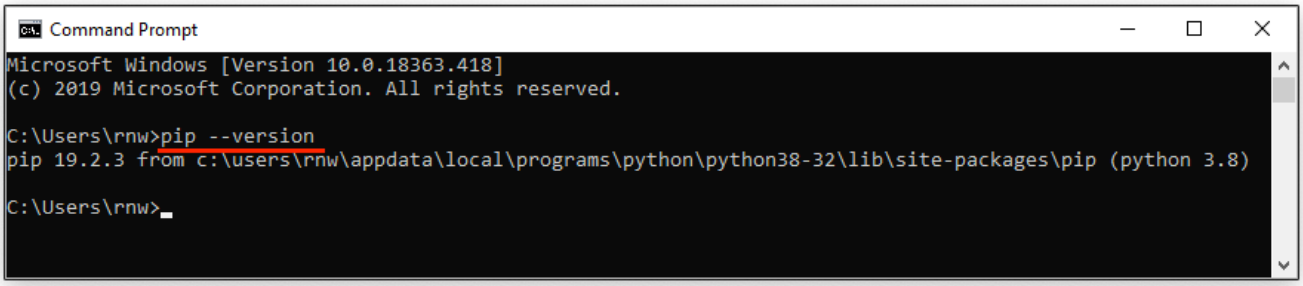
STEP - 2

Now, write a command to know version of pip:

pip --version

And press **Enter**.

You will see a version of installed pip.



Download & Install a Package using PIP:

STEP - 1

To download and **install** any python package, use following syntax:

Syntax – For Windows

```
pip install package_name
```

Syntax – For MacOS

```
pip3 install package_name
```

Syntax – For Linux

```
pip3 install package_name
```

STEP - 2

For example, To download and install **mysql-connector-python** module, use following command in **CMD**:

Example

```
pip install mysql-connector-python
```

STEP - 3

If you want to **remove** any python package, then use following syntax:

Syntax

```
pip uninstall package_name
```

Example

```
pip uninstall mysql-connector-python
```

Check all installed packages using PIP:

- To check all python packages installed on your system, use following command:

Example

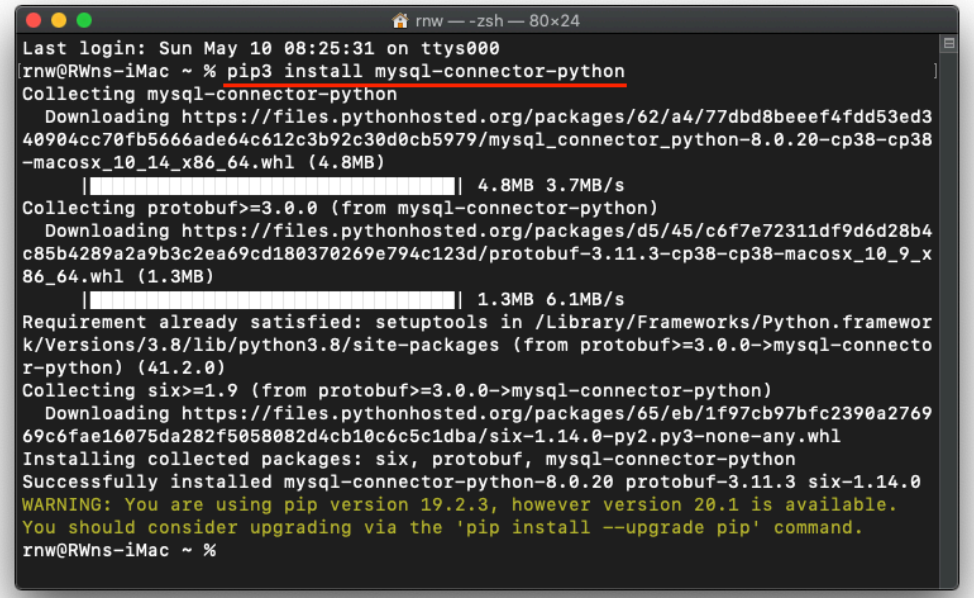
```
pip list

OR

pip freeze
```

SETTING UP mysql-connector-python MODULE

- To interact with **MySQL database**, a module called **mysql-connector-python** is used.
- For using that module, first of all we have to install it on our system.

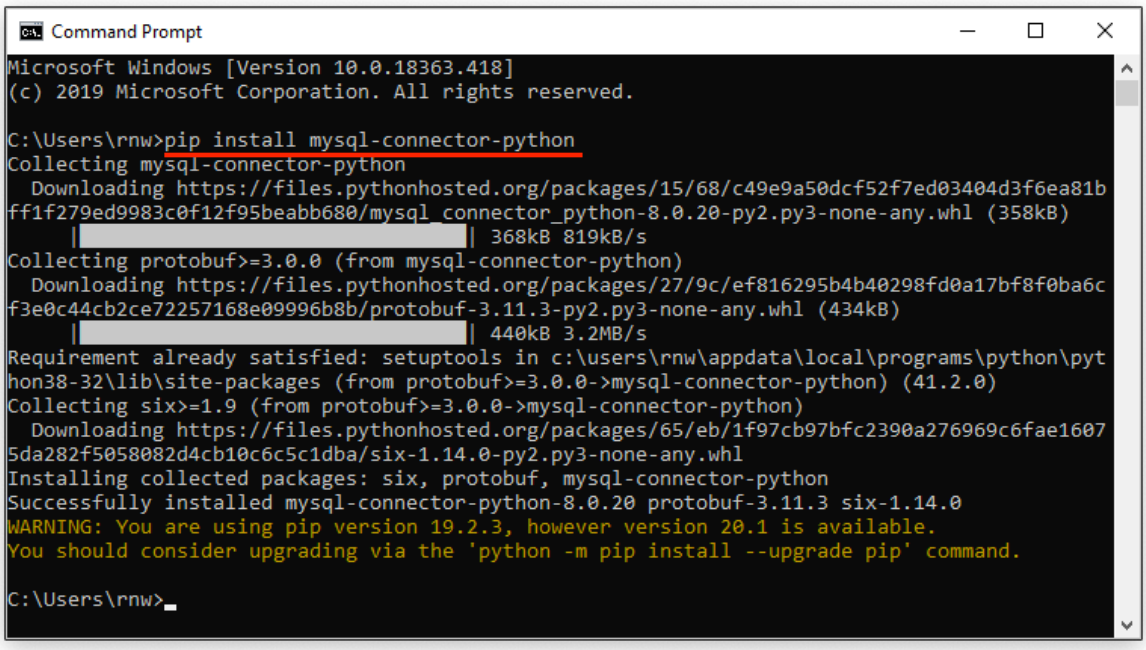
Installation on MacOS/Linux:

- Open Terminal, and enter following command: **pip3 install mysql-connector-python**
- Then, Press **Enter**.



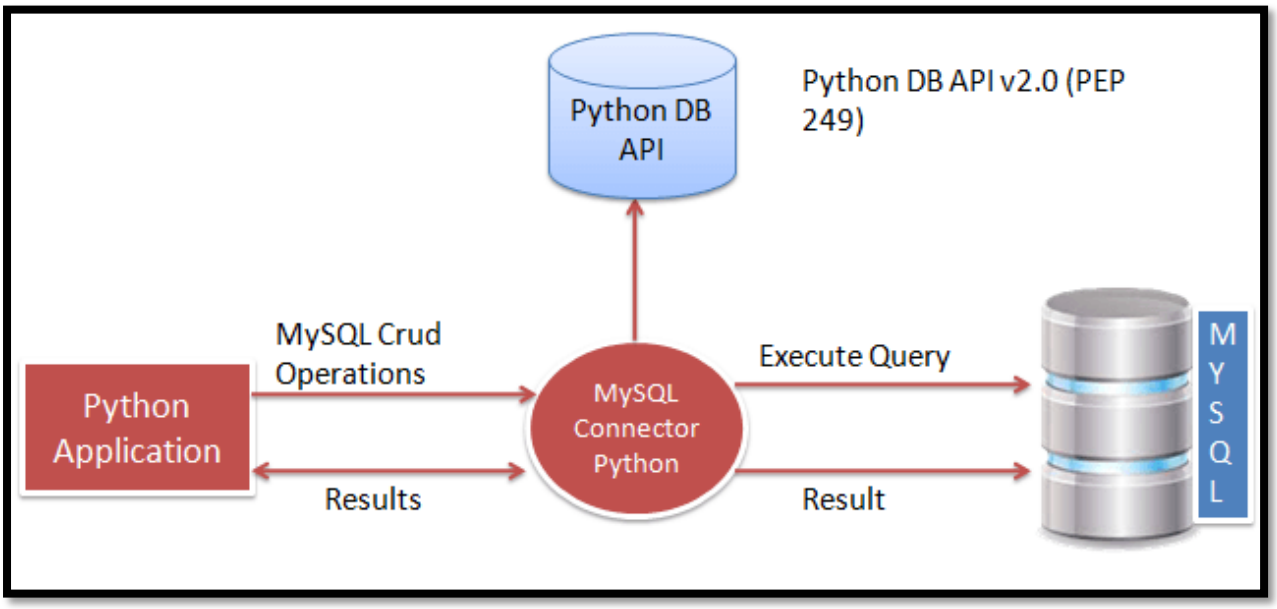
Installation on Windows:

- Open CMD, and enter following command: **pip install mysql-connector-python**
- Then, Press **Enter**.



CRUD OPERATION WITH mysql-connector-python MODULE

- Now, we can perform **CRUD operation** on MySQL database with Python using **mysql-connector-python** module.



- We will perform CRUD operations step-by-step as follows:
    - **MySQL Connection**
    - **Database Creation**
    - **Database Connection**
    - **Table Creation**
    - **Insert data into table**
    - **Retrieve data from table**
    - **Update data in table**
    - **Delete data from table**
    - **Delete Table**
    - **Delete Database**

A. MySQL Connection

- First, we create a connection with MySQL database.
- Arguments required to connect MySQL from Python with **mysql.connector.connect()** method:
  1. **host** - is the server name or Ip address on which MySQL is running. if you are running on localhost, then you can use localhost, or it's IP, i.e. 127.0.0.0
  2. **user** - i.e., the username that you use to work with MySQL Server. The default username for the MySQL database is a **root**
  3. **password** - Password is given by the user at the time of installing the MySQL database. If you are using root then you won't need the password.
  4. **database** - Database name to which you want to connect. This is **optional argument** and we will come back on this later.

Example

```
import mysql.connector as mysql
from mysql.connector import Error

try:
   conn = mysql.connect(host="localhost",user="root",password="")
   if conn.is_connected():
      print("Successfully Connected...")
except Error as e:
   print(e)
```

Output

```
Successfully Connected…
```

B. Database Creation

- After creating a connection with MySQL database, we execute a query which creates a database using **conn.cursor()** method.
- Use the connection object returned by a **connect()** method to create a **cursor** object to perform Database Operations.
- The **cursor.execute()** to execute SQL queries from Python.
- Close the Cursor object using a **cursor.close()** and MySQL database connection using **connection.close()** after your work completes.
- Catch Exception if any that may occur during this process.
- We going to create a database called **python_db**.

Example

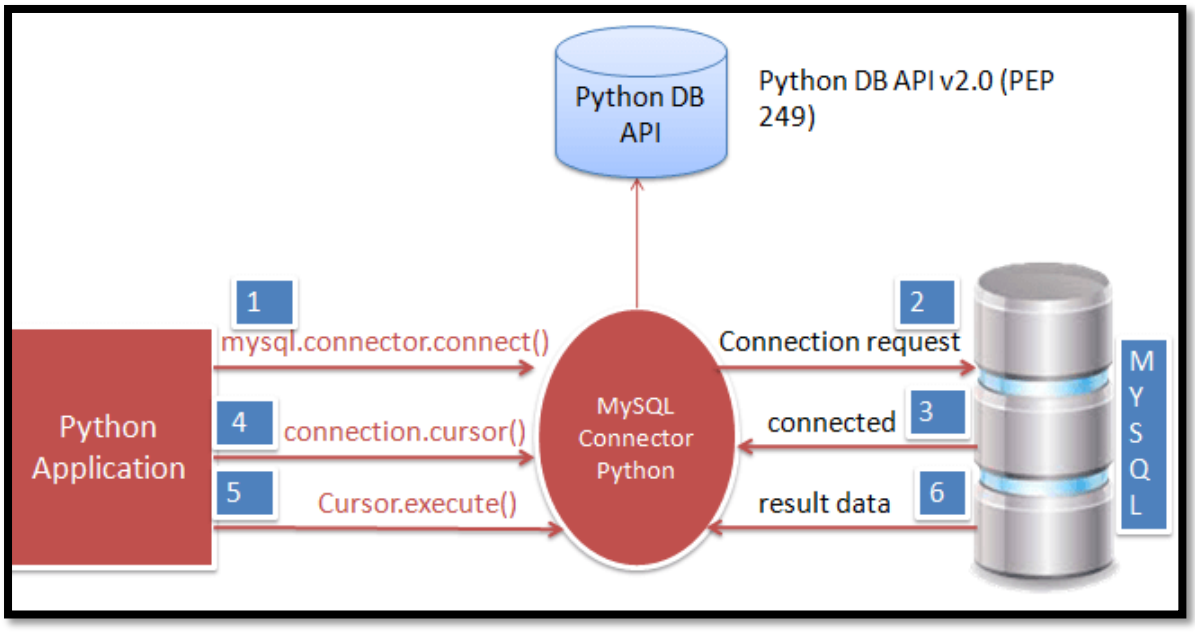
```
import mysql.connector as mysql
from mysql.connector import Error

try:
   conn = mysql.connect(host="localhost",user="root",password="")
   if conn.is_connected():
      cursor = conn.cursor()
      cursor.execute("CREATE DATABASE python_db;")
      print("Database Successfully Created...")
except Error as e:
   print("Error: ", e)
finally:
   if conn.is_connected():
      cursor.close()
      conn.close()
      print("MySQL connection is closed...")
```

Output

```
Database Successfully Created...
MySQL connection is closed...
```

C. Database Connection

- After creating our database, we now connecting to that database with help of **connect()** method as we saw earlier but now with all four arguments including **database**.



Example

```
import mysql.connector as mysql
from mysql.connector import Error

try:
   conn = mysql.connect(host="localhost",user="root",password="", database="python_db")
   if conn.is_connected():
      print("Database Successfully Connected...")
except Error as e:
   print("Error: ", e)
finally:
   if conn.is_connected():
      conn.close()
      print("MySQL connection is closed...")
```

Output

```
Database Successfully Connected...
MySQL connection is closed...
```

D. Table Creation

- Now we create our first table called **users** using **execute()** method.

Example

```
import mysql.connector as mysql
from mysql.connector import Error

try:
   conn = mysql.connect(host="localhost",user="root",password="",database="python_db")
   if conn.is_connected():
      cursor = conn.cursor()
      query = """CREATE TABLE users(
               id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
               name VARCHAR(30) NOT NULL,
               age INT(2) NOT NULL,
               city VARCHAR(20) NOT NULL);"""
      cursor.execute(query)
      print("Table Successfully Created...")
```

```
except Error as e:
    print("Error: ", e)
finally:
    if conn.is_connected():
        cursor.close()
        conn.close()
        print("MySQL connection is closed...")
```

Output

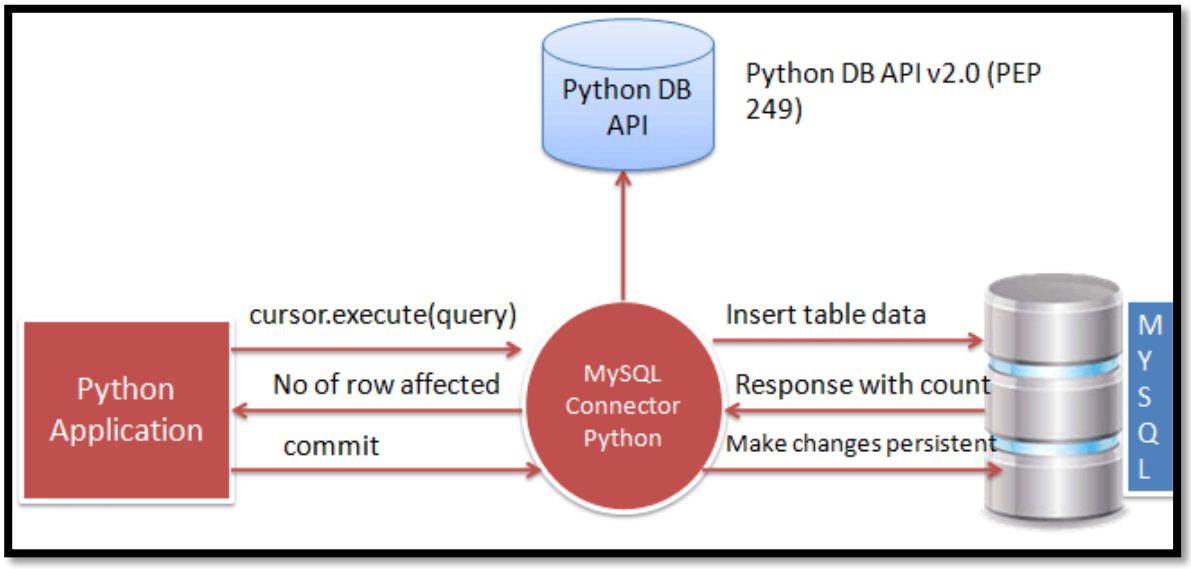
```
Table Successfully Created...
MySQL connection is closed...
```

E. Insert data into table

- Now we can insert our data or records into **users** table using **INSERT** query.
- We can insert data into table with 3 different styles of INSERT query:
    - Simple query
    - Parameterized query
    - Prepared Statement
- All above queries can be used to insert **single** or **multiple** records.
- As of now, we show how to insert single record using Simple and Parameterized queries. But we insert our multiple records using only Prepared Statement.

NOTE:

- **After the successful execution of query, Don't forget to commit your changes to database.**



Insert single record using Simple query:

Example

```
import mysql.connector as mysql
from mysql.connector import Error

try:
   conn = mysql.connect(host="localhost",user="root",password="",database="python_db")
   if conn.is_connected():
      cursor = conn.cursor()
      query = "INSERT INTO users(name,age,city) VALUES('Rahul',20,'Surat');"
      cursor.execute(query)
      conn.commit()
      print("Record Successfully Inserted...")
except Error as e:
   print("Error: ", e)
finally:
   if conn.is_connected():
      cursor.close()
      conn.close()
      print("MySQL connection is closed...")
```

Output

```
Record Successfully Inserted...
MySQL connection is closed...
```

Insert single record using Parameterized query:

- Now let's use above example with change of new record and Parameterized query.
- While executing a parameterized query, **execute()** method needs an extra argument of **tuple** consisting **values** to be placed into query.

Example

```
query = "INSERT INTO users(name,age,city) VALUES(%s,%s,%s);"
values = ('Viraj',18,'Mumbai')
cursor.execute(query, values)
```

Insert multiple records using Prepared Statement:

- Let's now insert multiple records at once using Prepared Statement.
- Prepared Statement is nothing but a Parameterized query.
- A cursor instantiated from **connection.cursor(prepared=True)** (MySQLCursorPrepared class) works like this:
- The first time you pass a SQL query statement to the cursor's execute() method, it prepares the statement.
- **For subsequent invocations of executing, the preparation phase is skipped if the SQL statement is the same**, i.e. the **query is not compiled again**.
- in the **first** cursor.execute(sql_insert_query, insert_tuple_1) **Python prepares statement** i.e. **Query gets compiled**.
- For **subsequent calls** of cursor.execute(sql_insert_query, insert_tuple_2) The query **will not be compiled again**, this step is **skipped and the query executed directly** with passed parameter values.

Example

```
import mysql.connector as mysql
from mysql.connector import Error

try:
   conn = mysql.connect(host="localhost",user="root",password="",database="python_db")
   if conn.is_connected():
      cursor = conn.cursor(prepared=True)
      query = "INSERT INTO users(name,age,city) VALUES(%s,%s,%s);"
      values1 = ('Raj',22,'Ahmedabad')
      values2 = ('Parth',20,'Ahmedabad')
      values3 = ('Manish',26,'Surat')
      cursor.execute(query, values1)
      cursor.execute(query, values2)
      cursor.execute(query, values3)
      conn.commit()
      print("All Records Successfully Inserted...")
except Error as e:
   print("Error: ", e)
finally:
   if conn.is_connected():
      cursor.close()
      conn.close()
      print("MySQL connection is closed...")
```

Output

```
All Records Successfully Inserted...
MySQL connection is closed...
```

- As an alternative of above example, we can create a **list of tuples** which contains our records values. And using **executemany()** method we can insert multiple records also.

Example

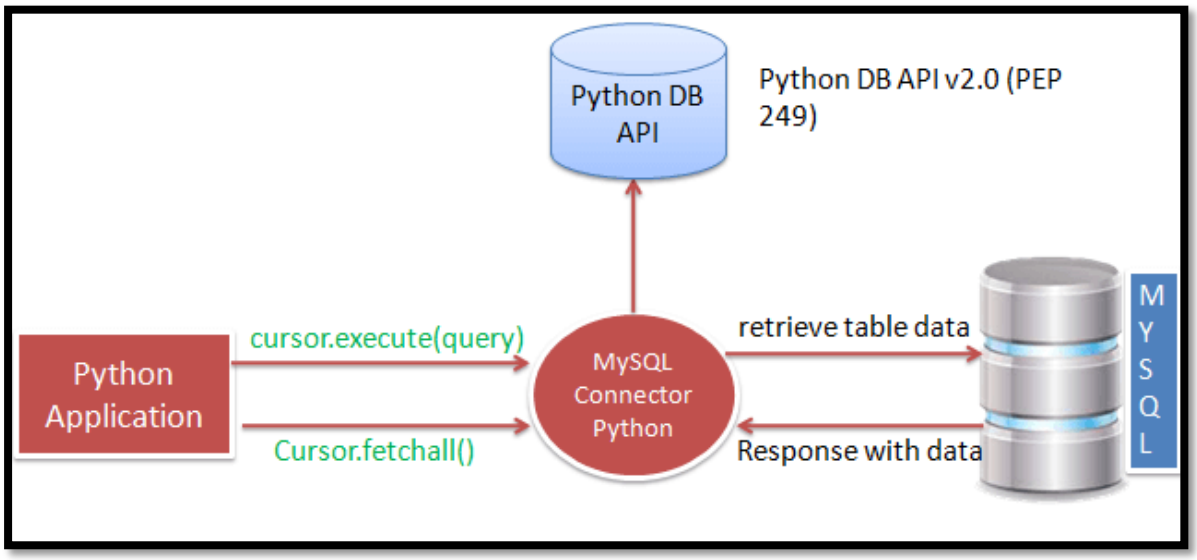
```
import mysql.connector as mysql
from mysql.connector import Error

try:
   conn = mysql.connect(host="localhost",user="root",password="",database="python_db")
   if conn.is_connected():
      cursor = conn.cursor(prepared=True)
      query = "INSERT INTO users(name,age,city) VALUES(%s,%s,%s);"
      values = [ ('Pratik',26,'Baroda'),
                 ('Yogesh',25,'Mumbai'),
                 ('Harsh',21,'Baroda') ]
      cursor.executemany(query, values)
      conn.commit()
      print("All Records Successfully Inserted...")
except Error as e:
   print("Error: ", e)
finally:
   if conn.is_connected():
      cursor.close()
      conn.close()
      print("MySQL connection is closed...")
```

Output

```
All Records Successfully Inserted...
MySQL connection is closed...
```

F. Retrieve data from table

- Now we can fetch records from table using **SELECT** query.
- For that, first we execute the **SELECT** query using the **cursor.execute()** method.
- Then, get **ResultSet** from the cursor object using a **cursor.fetchall()** method.
- Iterate over the ResultSet and **get each row** and its column value.



- We can fetch data using following methods:
  a. **cursor.fetchall() - To fetch all rows**
  b. **cursor.fetchone() - To fetch single row**
  c. **cursor.fetchmany(SIZE) - To fetch limited rows**

Fetch all rows using cursor.fetchall():

Example

```
cursor = conn.cursor()
query = "SELECT * FROM users;"
cursor.execute(query)
res = cursor.fetchall()
print("Total records: ", cursor.rowcount)              # cursor.rowcount returns no. of rows
for row in res:
    print(f"ID: {row[0]} Name: {row[1]} Age: {row[2]} City: {row[3]}")
print("All Records Successfully Retrieved...")
```

Output

```
Total records:  8
ID: 2 Name: Rahul Age: 20 City: Surat
ID: 3 Name: Viraj Age: 18 City: Mumbai
ID: 4 Name: Raj Age: 22 City: Ahmedabad
ID: 5 Name: Parth Age: 20 City: Ahmedabad
ID: 6 Name: Manish Age: 26 City: Surat
ID: 7 Name: Pratik Age: 26 City: Baroda
ID: 8 Name: Yogesh Age: 25 City: Mumbai
ID: 9 Name: Harsh Age: 21 City: Baroda
All Records Successfully Retrieved...
MySQL connection is closed...
```

Fetch single row using cursor.fetchone():

- We can fetch data using following methods:
- Use the **cursor.fetchone()** method to **retrieve the next row** of a query result set.
- This method returns a **single record** or **None** if no more rows are available.

Example

```
import mysql.connector as mysql
from mysql.connector import Error

try:
    conn = mysql.connect(host="localhost",user="root",password="",database="python_db")
    if conn.is_connected():
        cursor = conn.cursor()
        query = "SELECT * FROM users;"
        cursor.execute(query)
        res = cursor.fetchone()
        print(res)
        print("Record Successfully Retrieved...")
except Error as e:
    print("Error: ", e)
finally:
    if conn.is_connected():
        cursor.close()
        conn.close()
        print("MySQL connection is closed...")
```

Output

```
(2, 'Rahul', 20, 'Surat')
Record Successfully Retrieved...
```

- In above output, we don't get print **MySQL connection is closed…** Why? Because **conn.cursor()** method raise **MySQL Unread result error** internally. It thinks that now just one row fetched, and until all rows will not fetched it keep connection open with MySQL.

NOTE:

- **We have to set buffered=True in conn.cursor() method to avoid MySQL Unread result error.**

Example

```
import mysql.connector as mysql
from mysql.connector import Error

try:
   conn = mysql.connect(host="localhost",user="root",password="",database="python_db")
   if conn.is_connected():
      cursor = conn.cursor(buffered=True)
      query = "SELECT * FROM users;"
      cursor.execute(query)
      res = cursor.fetchone()
      print(res)
      print("Record Successfully Retrieved...")
except Error as e:
   print("Error: ", e)
finally:
   if conn.is_connected():
      cursor.close()
      conn.close()
      print("MySQL connection is closed...")
```

Output

```
(2, 'Rahul', 20, 'Surat')
Record Successfully Retrieved...
MySQL connection is closed...
```

Fetch limited rows using cursor.fetchmany(SIZE):

- This method fetches the next set of rows of a query result and returns a **list of tuples**. **If no more rows are available, it returns an empty list.**
- Cursor's **fetchmany()** methods return the **number of rows specified by size argument**, **defaults value** of **size** argument is **one**. If the specified size is 5, then it returns Five rows.

NOTE:

- **If a table contains row lesser than specified size then fewer rows are returned.**

Example

```
cursor = conn.cursor(buffered=True)
query = "SELECT * FROM users;"
cursor.execute(query)
res = cursor.fetchmany(size=4)
print(res)
print("Record Successfully Retrieved...")
```

Output

```
[(2, 'Rahul', 20, 'Surat'), (3, 'Viraj', 18, 'Mumbai'), (4, 'Raj', 22, 'Ahmedabad'), (5, 'Parth', 20, 'Ahmedabad')]
Record Successfully Retrieved...
MySQL connection is closed...
```

- **Default value** of **size** argument is **one.**

Example

```
cursor = conn.cursor(buffered=True)
query = "SELECT * FROM users;"
cursor.execute(query)
res = cursor.fetchmany()
print(res)
print("Record Successfully Retrieved...")
```

Output

```
[(2, 'Rahul', 20, 'Surat')]
Record Successfully Retrieved...
MySQL connection is closed...
```

- We can also fetch MySQL data using the **column names** instead of a column id.
- For that, we just have to pass an argument **dictionary=True** to conn.cursor() method. Now **MySQLCursorDict** creates a cursor that returns **rows as dictionaries** so we can access using **column name** (here column name is the **key** of the dictionary).

Example

```
cursor = conn.cursor(dictionary=True)
    query = "SELECT * FROM users;"
    cursor.execute(query)
    res = cursor.fetchall()
    print("Total records: ", cursor.rowcount)
    for row in res:
        print(f"ID: {row['id']} Name: {row['name']} Age: {row['age']} City: {row['city']}")
    print("All Records Successfully Retrieved...")
```

Output

```
Total records:  8
ID: 2 Name: Rahul Age: 20 City: Surat
ID: 3 Name: Viraj Age: 18 City: Mumbai
ID: 4 Name: Raj Age: 22 City: Ahmedabad
ID: 5 Name: Parth Age: 20 City: Ahmedabad
ID: 6 Name: Manish Age: 26 City: Surat
ID: 7 Name: Pratik Age: 26 City: Baroda
ID: 8 Name: Yogesh Age: 25 City: Mumbai
ID: 9 Name: Harsh Age: 21 City: Baroda
All Records Successfully Retrieved...
MySQL connection is closed...
```

G. Update data in table

- We can update records in table using **UPDATE** query.

Example

```
import mysql.connector as mysql
from mysql.connector import Error

try:
   conn = mysql.connect(host="localhost",user="root",password="",database="python_db")
   if conn.is_connected():
      cursor = conn.cursor(dictionary=True)
      select_query = "SELECT * FROM users WHERE id=%s;"
      uid = (3,)
      cursor.execute(select_query,uid)
      res = cursor.fetchone()
      print(res)

      update_query = "UPDATE users SET name=%s WHERE id=%s;"
      values = ('Yash',uid[0])
      cursor.execute(update_query,values)
      conn.commit()
      print("Record Successfully Updated...")

      print("After updating record: ")
      cursor.execute(select_query,uid)
      res = cursor.fetchone()
      print(res)
except Error as e:
   print("Error: ", e)
finally:
   if conn.is_connected():
      cursor.close()
      conn.close()
      print("MySQL connection is closed...")
```

Output

```
{'id': 3, 'name': 'Viraj', 'age': 18, 'city': 'Mumbai'}
Record Successfully Updated...
After updating record:
{'id': 3, 'name': 'Yash', 'age': 18, 'city': 'Mumbai'}
MySQL connection is closed...
```

NOTE:

- **After the successful execution of query, Don't forget to commit your changes to database.**

- Update multiple records in a single query:

Example

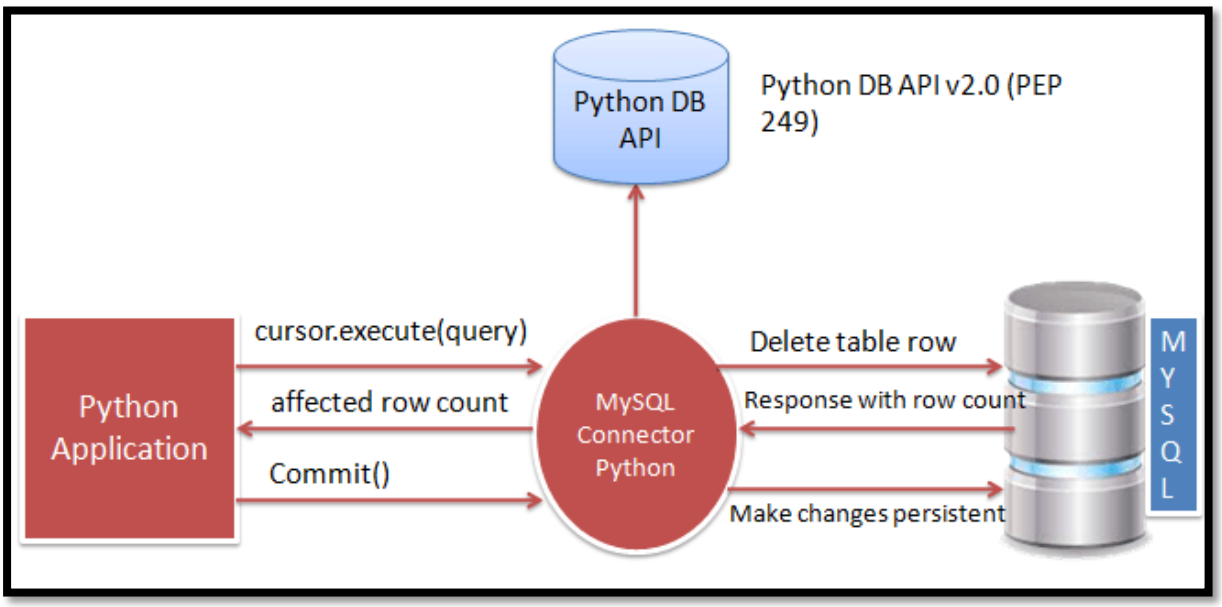
```
cursor = conn.cursor()
query = "UPDATE users SET age=%s, city=%s WHERE id=%s;"
values = [(30, 'Mumbai', 4), (24, 'Baroda', 2)]
cursor.executemany(query, values)
conn.commit()
print(cursor.rowcount, " records updated successfully...")
```

Output

```
2  records updated successfully...
MySQL connection is closed...
```

H. Delete data from table

- We can delete rows from table using **DELETE** query and delete columns using **ALTER** query.



- We see all scenarios about deleting data as follows:
  a. Delete single row
  b. Delete multiple rows
  c. Delete all rows
  d. Delete single column
  e. Delete multiple columns

Delete single row:

Example

```
import mysql.connector as mysql
from mysql.connector import Error

try:
   conn = mysql.connect(host="localhost",user="root",password="",database="python_db")
   if conn.is_connected():
      cursor = conn.cursor()
      query = "DELETE FROM users WHERE id=%s;"
      value = (3,)
      cursor.execute(query, value)
      conn.commit()
      print("Record Deleted Successfully...")
except Error as e:
   print("Error: ", e)
finally:
   if conn.is_connected():
      cursor.close()
      conn.close()
      print("MySQL connection is closed...")
```

Output

```
Record Deleted Successfully...
MySQL connection is closed...
```

Delete multiple rows:

Example

```
cursor = conn.cursor()
query = "DELETE FROM users WHERE id=%s;"
values = [ (6,), (7,) ]
cursor.executemany(query, values)
conn.commit()
print(cursor.rowcount, "Records Deleted Successfully...")
```

Output

```
2 Records Deleted Successfully...
MySQL connection is closed...
```

Delete all rows:

Example

```
cursor = conn.cursor()
query = "DELETE FROM users;"
cursor.execute(query)
conn.commit()
print("All Records Deleted Successfully...")
```

Output

```
All Records Deleted Successfully...
MySQL connection is closed...
```

Delete single column:

Example

```
cursor = conn.cursor()
query = "ALTER TABLE users DROP COLUMN age;"
cursor.execute(query)
conn.commit()
print("Column Deleted Successfully...")
```

Output

```
Column Deleted Successfully...
MySQL connection is closed...
```

Delete multiple columns:

Example

```
cursor = conn.cursor()
query = "ALTER TABLE users DROP COLUMN name, DROP COLUMN city;"
cursor.execute(query)
conn.commit()
print("Columns Deleted Successfully...")
```

Output

```
Columns Deleted Successfully...
MySQL connection is closed...
```

I. Delete Table

- We can delete old and unused tables from using **DROP TABLE** query.

Example

```
cursor = conn.cursor()
query = "DROP TABLE users;"
cursor.execute(query)
conn.commit()
print("Table Deleted Successfully...")
```

Output

```
Table Deleted Successfully...
MySQL connection is closed...
```

J. Delete Database

- We can delete old and unused database from using **DROP DATABSE** query.

Example

```
cursor = conn.cursor()
query = "DROP DATABASE python_db;"
cursor.execute(query)
conn.commit()
print("Database Deleted Successfully...")
```

Output

```
Database Deleted Successfully...
MySQL connection is closed...
```

#12 GUI WITH TKINTER

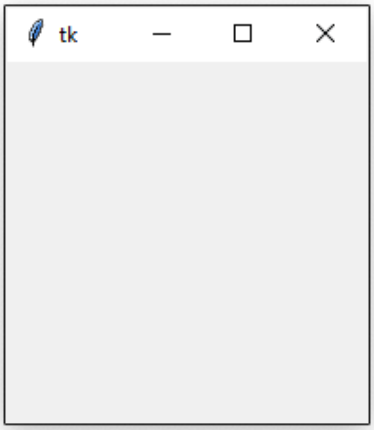
tkinter MODULE

- Python provides the standard library **Tkinter** for creating the **graphical user interface (GUI)** for desktop based applications.
- Developing desktop based applications with python Tkinter is not a complex task. An empty Tkinter top-level window can be created by using the following steps.
  1. **import the Tkinter module.**
  2. **Create the main application window.**
  3. **Add the widgets like labels, buttons, frames, etc. to the window.**
  4. **Call the main event loop so that the actions can take place on the user's computer screen.**

Example

```
import tkinter

# creating application main window
root = tkinter.Tk()

# entering the event main loop
root.mainloop()
```

Output



tkinter WIDGETS

- There are various **widgets** like Button, Frame, Label, entry, etc. that are used to build the python GUI applications.

No.	Widget	Description
1	**Button**	The Button is used to add various kinds of buttons to the python application.
2	**Canvas**	The canvas widget is used to draw the canvas on the window.
3	**Checkbutton**	The Checkbutton is used to display the CheckButton on the window.
4	**Entry**	The entry widget is used to display the single-line text field to the user. It is commonly used to accept user values.
5	**Frame**	It can be defined as a container to which, another widget can be added and organized.
6	**Label**	A label is a text used to display some message or information about the other widgets.
7	**ListBox**	The ListBox widget is used to display a list of options to the user.
8	**Menubutton**	The Menubutton is used to display the menu items to the user.
9	**Menu**	It is used to add menu items to the user.
10	**Message**	The Message widget is used to display the message-box to the user.
11	**Radiobutton**	The Radiobutton is different from a checkbutton. Here, the user is provided with various options and the user can select only one option among them.
12	**Scale**	It is used to provide the slider to the user.
13	**Scrollbar**	It provides the scrollbar to the user so that the user can scroll the window up and down.
14	**Text**	It is different from Entry because it provides a multi-line text field to the user so that the user can write the text and edit the text inside it.
15	**Toplevel**	It is used to create a separate window container.
16	**Spinbox**	It is an entry widget used to select from options of values.
17	**PanedWindow**	It is like a container widget that contains horizontal or vertical panes.

18	**LabelFrame**	A LabelFrame is a container widget that acts as the container
19	**MessageBox**	This module is used to display the message-box in the desktop based applications.
20	**PhotoImage**	The PhotoImage class is used to display images (either grayscale or true color images) in labels, buttons, canvases, and text widgets.

tkinter GEOMETRY

- The Tkinter geometry specifies the method by using which, the **widgets are represented on display**.
- The Python Tkinter provides the following geometry methods:
  - **A.** pack()
  - **B.** grid()
  - **C.** place()

A. pack() method

- The **pack()** widget is used to **organize widget in the block**.
- The positions widgets added to the python application using the pack() method can be controlled by using the various options specified in the method call.
- However, the controls are less and widgets are generally added in the less organized manner.

Syntax

```
widgets.pack(options)
```

- A list of possible options that can be passed in pack() is given below:
  - **expand** - The expand option tells the manager to assign additional space to the widget box. If the parent widget is made larger than necessary to hold all packed widgets, any exceeding space will be distributed among all widgets that have the expand option set to a **non-zero value**.
  - **fill - By default**, the fill is set to **NONE**. However, we can set it to X, Y or BOTH to determine

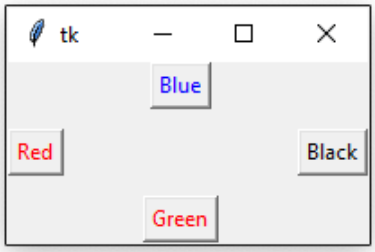
whether the widget contains any extra space. **BOTH** means that the widget should expand both **horizontally and vertically**, **X** means that it should expand only **horizontally**, and **Y** means that it should expand only **vertically**.

- **side -** it represents the side of the parent to which the widget is to be placed on the window.

Example

```
from tkinter import *

root = Tk()
root.geometry("200x100")

redbutton = Button(root, text = "Red", fg = "red")
redbutton.pack( side = LEFT)

greenbutton = Button(root, text = "Black", fg = "black")
greenbutton.pack( side = RIGHT )

bluebutton = Button(root, text = "Blue", fg = "blue")
bluebutton.pack( side = TOP )

blackbutton = Button(root, text = "Green", fg = "red")
blackbutton.pack( side = BOTTOM)

root.mainloop()
```

Output

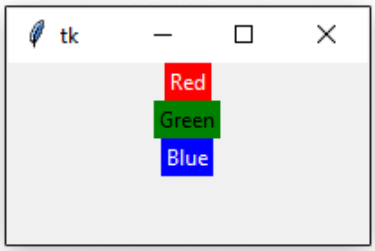


- To put a number of widgets in a **column**, you can use the pack method without any options:

Example

```
from tkinter import *

root = Tk()
root.geometry("200x100")

w = Label(root, text="Red", bg="red", fg="white")
w.pack()
w = Label(root, text="Green", bg="green", fg="black")
w.pack()
w = Label(root, text="Blue", bg="blue", fg="white")
w.pack()

root.mainloop()
```

Output



- You can use the **fill=X** option to make all widgets as wide as the parent widget:

Example

```
from tkinter import *

root = Tk()
root.geometry("200x100")

w = Label(root, text="Red", bg="red", fg="white")
w.pack(fill=X)
w = Label(root, text="Green", bg="green", fg="black")
w.pack(fill=X)
w = Label(root, text="Blue", bg="blue", fg="white")
w.pack(fill=X)

root.mainloop()
```

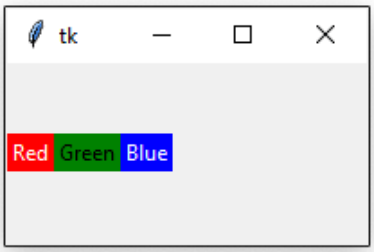
Output

- To pack widgets side by side, use the side option. If you wish to make the widgets as high as the parent, use the **fill=Y** option too:

Example

```
from tkinter import *

root = Tk()
root.geometry("200x100")

w = Label(root, text="Red", bg="red", fg="white")
w.pack(side=LEFT)
w = Label(root, text="Green", bg="green", fg="black")
w.pack(side=LEFT)
w = Label(root, text="Blue", bg="blue", fg="white")
w.pack(side=LEFT)

root.mainloop()
```

Output



B. grid() method

- The **grid()** geometry manager organizes the widgets in the **tabular form**.
- We can specify the **rows** and **columns** as the options in the method call. We can also specify the column span (width) or rowspan (height) of a widget.
- This is a more organized way to place the widgets to the python application.

Syntax

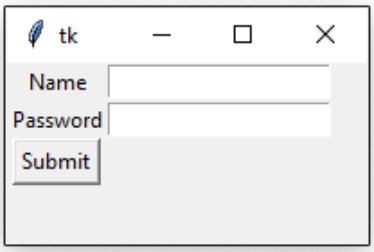
```
widgets.grid(options)
```

- A list of possible options that can be passed in grid() is given below:
  - **column** - The column number in which the widget is to be placed. The leftmost column is represented by 0.
  - **columnspan** - The width of the widget. It represents the number of columns up to which, the column is expanded.
  - **ipadx, ipady** - It represents the number of pixels to pad the widget inside the widget's border.
  - **padx, pady** - It represents the number of pixels to pad the widget outside the widget's border.
  - **row** - The row number in which the widget is to be placed. The topmost row is represented by 0.
  - **rowspan** - The height of the widget, i.e. the number of the row up to which the widget is expanded.
  - **sticky** - If the cell is larger than a widget, then sticky is used to specify the position of the widget inside the cell. It may be the concatenation of the sticky letters representing the position of the widget. It may be **N, E, W, S, NE, NW, NS, EW, ES**. Here, **W** means **West**.

Example

```
from tkinter import *

root = Tk()
root.geometry("200x100")

name = Label(root,text = "Name").grid(row = 0, column = 0)
e1 = Entry(root).grid(row = 0, column = 1)
password = Label(root,text = "Password").grid(row = 1, column = 0)
e2 = Entry(root).grid(row = 1, column = 1)
submit = Button(root, text = "Submit").grid(row = 4, column = 0)

root.mainloop()
```

Output

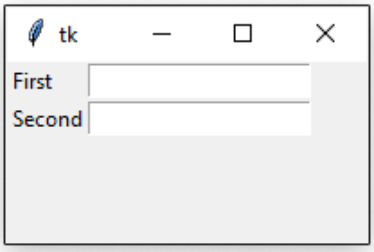


- Note that the widgets are centered in their cells. You can use the sticky option to change this:

Example

```
from tkinter import *

root = Tk()
root.geometry("200x100")

Label(root, text="First").grid(row=0, sticky=W)
Label(root, text="Second").grid(row=1, sticky=W)
Entry(root).grid(row=0, column=1)
Entry(root).grid(row=1, column=1)

root.mainloop()
```

Output



C. place() method

- The **place()** geometry manager **organizes** the widgets to the specific **x and y coordinates**.

Syntax

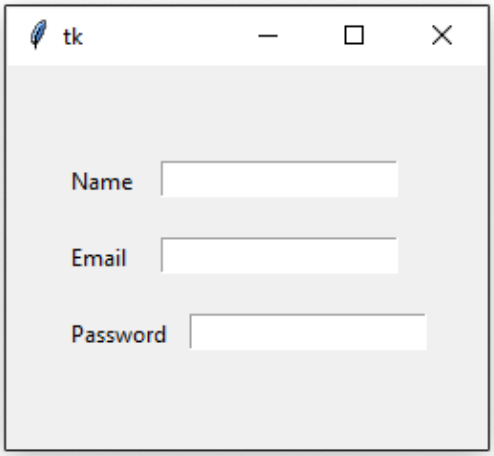
```
widgets.place(options)
```

- A list of possible options that can be passed in grid() is given below:
  - **anchor** - It represents the exact position of the widget within the container. The default value (direction) is NW (the upper left corner)
  - **bordermode** - The default value of the border type is INSIDE that refers to ignore the parent's inside the border. The other option is OUTSIDE.
  - **height, width** - It refers to the height and width in pixels.
  - **relheight, relwidth** - It is represented as the float between 0.0 and 1.0 indicating the fraction of the parent's height and width.
  - **relx, rely** - It is represented as the float between 0.0 and 1.0 that is the offset in the horizontal and vertical direction.
  - **x, y** - It refers to the horizontal and vertical offset in the pixels.

Example

```
from tkinter import *

root = Tk()
root.geometry("250x200")

name = Label(root, text = "Name").place(x = 30,y = 50)
email = Label(root, text = "Email").place(x = 30, y = 90)
password = Label(root, text = "Password").place(x = 30, y = 130)
e1 = Entry(root).place(x = 80, y = 50)
e2 = Entry(root).place(x = 80, y = 90)
e3 = Entry(root).place(x = 95, y = 130)

root.mainloop()
```

Output



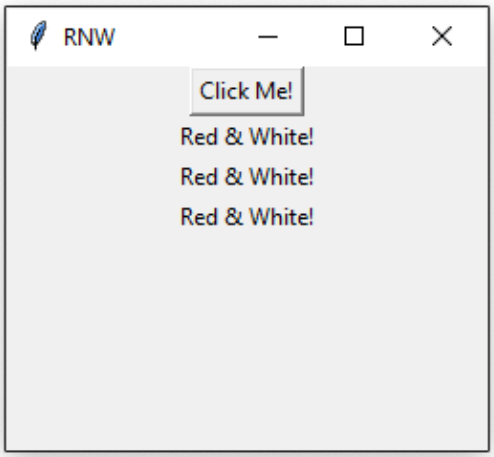
BINDING FUNCTIONS

- **Binding** or **Command functions** are those who are called **whenever an event occurs or is triggered**.
- We can bind any function to **Button** widget with **command** argument.

Example

```
from tkinter import *

root = Tk()
root.geometry("250x200")
root.title("RNW")                # to change title of our GUI window

def rnw():
   Label(root, text="Red & White!").pack()

Button(root, text="Click Me!", command=rnw).pack()

root.mainloop()
```

Output



MOUSE CLICKING EVENTS

- The **bind** **method** provides you with a very simplistic approach to implementing the mouse clicking events.
- Let's look at the three pre-defined functions which can be used out of the box with the bind method.
- Clicking events are of three types **leftClick**, **middleClick**, and **rightClick**.
  - **<Button-1>** parameter of the bind method is the **left-clicking event**, i.e., when you click the left button, the bind method will call the function specified as a second parameter to it.
  - **<Button-2>** for **middle-click**
  - **<Button-3>** for **right-click**

- We can bind any function to almost any widget with **bind** **method**.

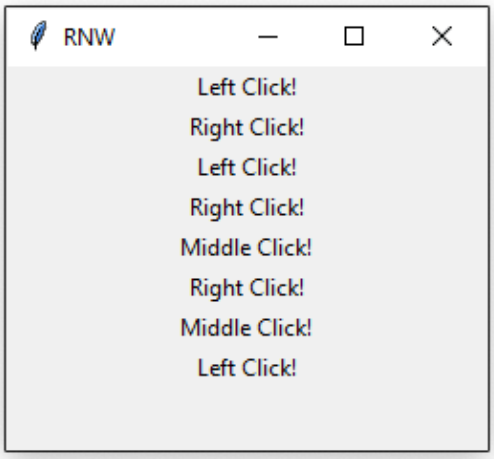
Syntax

```
widgets.bind("<Button-1/2/3>", function)
```

Example

```
from tkinter import *

root = Tk()
root.geometry("250x200")
root.title("RNW")

def left_click(event):
   Label(root, text = "Left Click!").pack()

def middle_click(event):
   Label(root, text = "Middle Click!").pack()

def right_click(event):
   Label(root, text = "Right Click!").pack()

root.bind("<Button-1>", left_click)
root.bind("<Button-2>", middle_click)
root.bind("<Button-3>", right_click)

root.mainloop()
```

Output



NOTE:

- **We must have to catch an event in function definition, because when a widget gets clicked with bind method, that widget pass an event to that function.**

TYPES OF POP-UP BOXES USING messagebox()

- The **messagebox** module is used to display the message boxes in the python applications.
- There are the various functions which are used to display the relevant messages depending upon the application requirements.

Syntax

```
messagebox.function_name(title, message)
```

- Parameters:
  - **function_name -** It represents an appropriate message box function.
  - **title -** It is a string which is shown as a title of a message box.
  - **message -** It is the string to be displayed as a message on the message box.

- There is one of the following functions used to show the appropriate message boxes. All the functions are used with the same syntax but have the specific functionalities.
  A. **showinfo()**
  B. **showwarning()**
  C. **showerror()**
  D. **askquestion()**
  E. **askokcancle()**
  F. **askyesno()**
  G. **askretrycancle()**

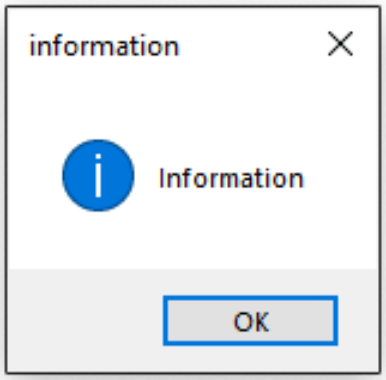
A. showinfo()

- The **showinfo()** messagebox is used where we need to **show some relevant information to the user**.

Example

```
from tkinter import *
from tkinter import messagebox
# mandatory to import separately because it is a module not a class

root = Tk()
messagebox.showinfo("information","Information")
root.mainloop()
```

Output



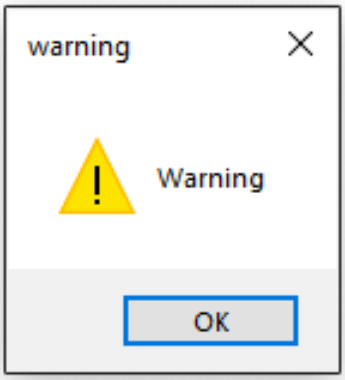
B. showwarning()

- This method is used to **display** the **warning to the user**.

Example

```
from tkinter import *
from tkinter import messagebox

root = Tk()

messagebox.showwarning("warning","Warning")

root.mainloop()
```

Output



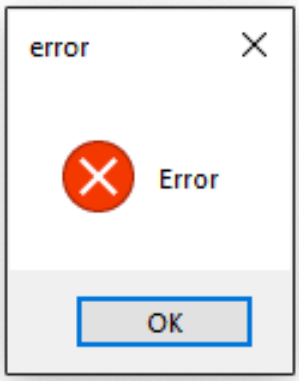
C. showerror()

- This method is used to **display** the **error message to the user**.

Example

```
from tkinter import *
from tkinter import messagebox

root = Tk()

messagebox.showerror("error","Error")

root.mainloop()
```

Output



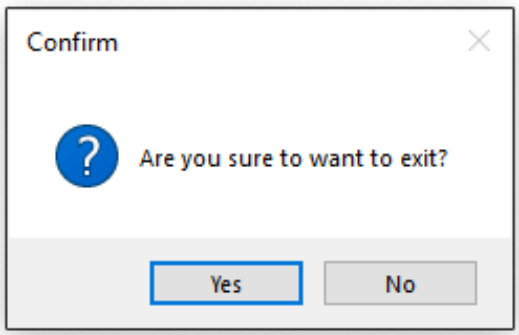
D. askquestion()

- This method is used to ask some question to the user which can be answered in **yes** or **no**.

Example

```
from tkinter import *
from tkinter import messagebox

root = Tk()

messagebox.askquestion("Confirm","Are you sure to want to exit?")

root.mainloop()
```

Output



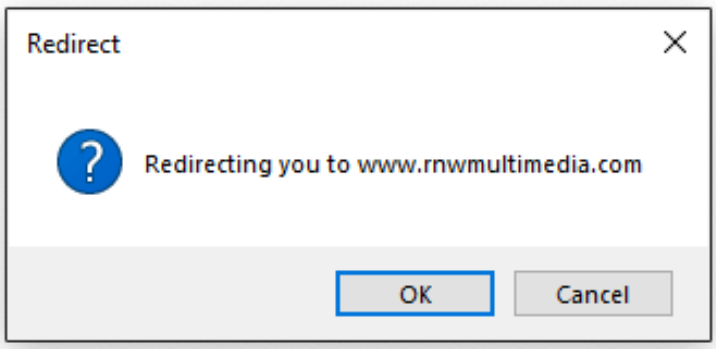
E. askokcancel()

- This method is used to **confirm the user's action** regarding some application activity.

Example

```
from tkinter import *
from tkinter import messagebox

root = Tk()

messagebox.askokcancel("Redirect","Redirecting you to www.rnwmultimedia.com")

root.mainloop()
```

Output



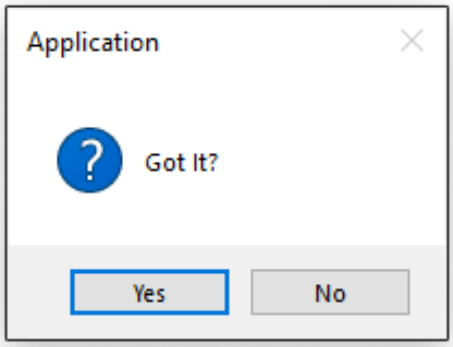
F. askyesno()

- This method is used to **ask the user about some action** to which, the user can answer in **yes** or **no**.

Example

```
from tkinter import *
from tkinter import messagebox

root = Tk()

messagebox.askyesno("Application","Got It?")

root.mainloop()
```

Output



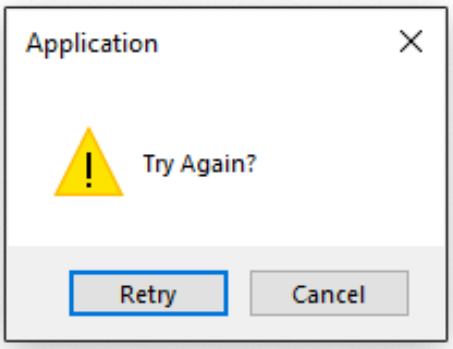
G. askretrycancel()

- This method is used to **ask the user about doing a particular task again or not**.

Example

```
from tkinter import *
from tkinter import messagebox

root = Tk()

messagebox.askretrycancel("Application","Try Again?")

root.mainloop()
```

Output



PhotoImage()

- You can use the **PhotoImage** class whenever you need to display an **icon** or an **image** in a Tkinter application.

NOTE:
- **PhotoImage class can only display images which are in PNG, GIF, PGM/PPM format. To display images which have another format, we will have to use pillow library.**

Syntax

```
PhotoImage(data, format, file, gamma, height, palette, width)
```

- In PhotoImage class, all arguments are optional except **file** or **data**.
- To display above mentioned format contained images, use **file** argument.
- The PhotoImage can also read **base64-encoded GIF** files from **strings**. You can use this to embed images in Python source code (use functions in the base64 module to convert binary data to base64-encoded strings), for that use **data** argument:

Example

```
from tkinter import *

root = Tk()
root.title("RNW")

icon = PhotoImage(file="rnw.png")
label = Label(root, image=icon)
label.pack()

root.mainloop()
```

Output

DISPLAY IMAGE USING pillow LIBRARY

- If you need to work with other file formats, the **Python Imaging Library (PIL)** contains classes that lets you load images in over 30 formats, and convert them to Tkinter-compatible image objects.
- For using PIL, first you have to install **pillow** library. You can download and install it with help of pip:
  pip install pillow

Example

```
from tkinter import *
from PIL import Image, ImageTk

root = Tk()
root.title("RNW Ruby")

image = Image.open("ruby.jpg")
photo = ImageTk.PhotoImage(image)
label = Label(image=photo)
label.image = photo                    # keep a reference!
label.pack()

root.mainloop()
```

NOTE:

- When a **PhotoImage object is garbage-collected** by Python (e.g. when you return from a function which stored an image in a local variable), **the image is cleared even if it's being displayed by a Tkinter widget**.
- To avoid this, the program **must keep an extra reference to the image object**. A simple way to do this is to assign the image to a widget attribute, like this:

```
label = Label(image=photo)
label.image = photo                    # keep a reference!
label.pack()
```

Output