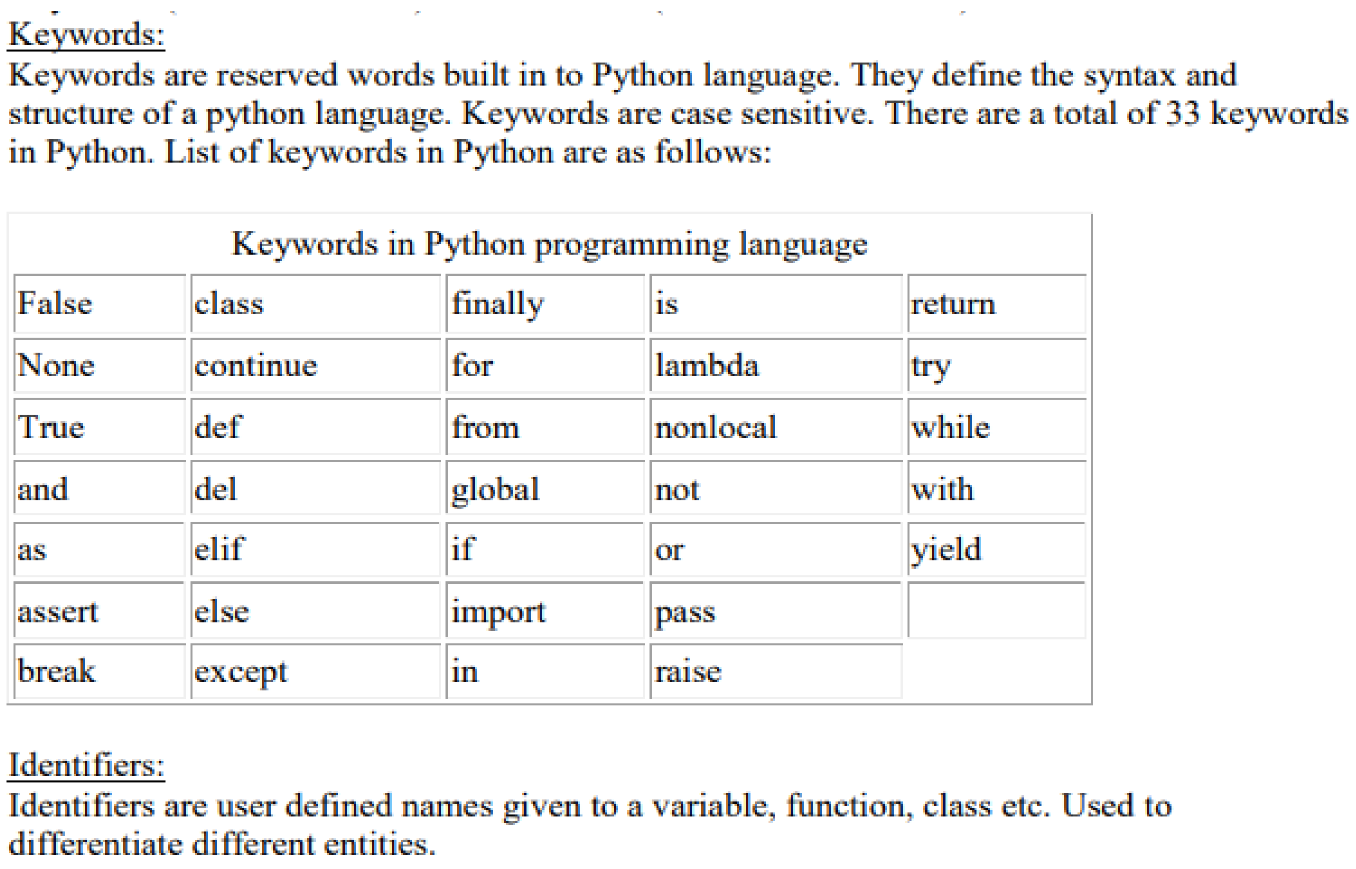
PYTHON

- ❖ Case sensitive.
- ❖ **#** – is used for single line commenting
- ❖ “”” “”” - is used for multi line commenting



Keywords:

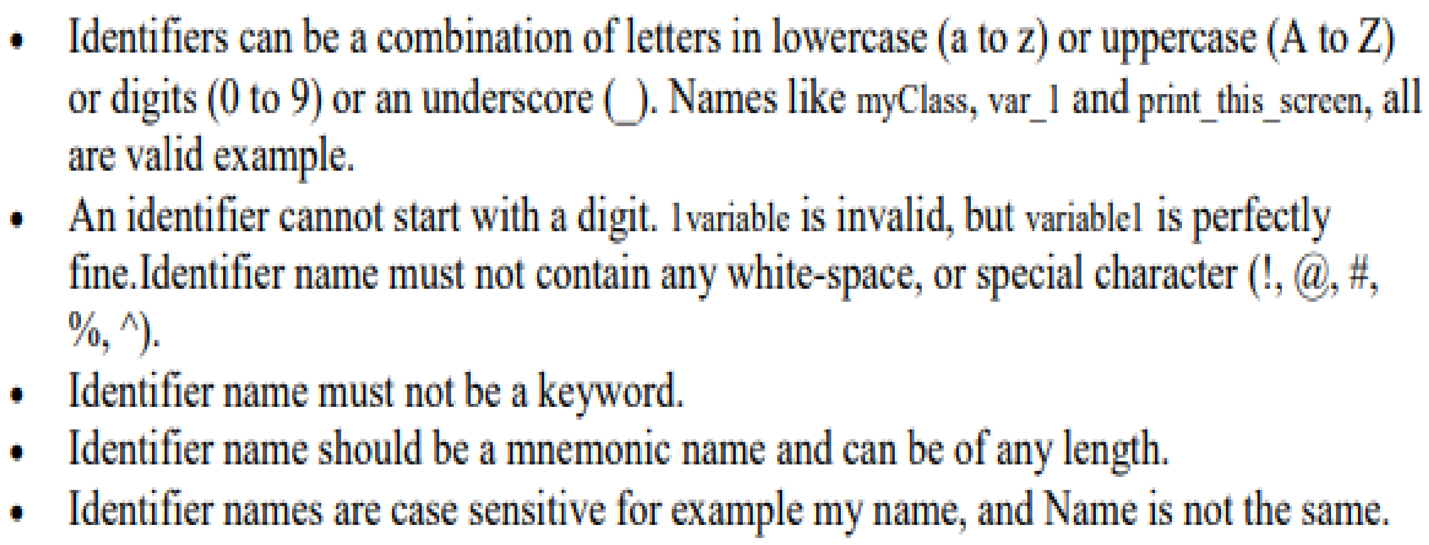
Keywords are reserved words built in to Python language. They define the syntax and structure of a python language. Keywords are case sensitive. There are a total of 33 keywords in Python. List of keywords in Python are as follows:

Keywords in Python programming language

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

Identifiers:

Identifiers are user defined names given to a variable, function, class etc. Used to differentiate different entities.



- Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). Names like myClass, var_1 and print_this_screen, all are valid example.
- An identifier cannot start with a digit. 1variable is invalid, but variable1 is perfectly fine.Identifier name must not contain any white-space, or special character (!, @, #, %, ^).
- Identifier name must not be a keyword.
- Identifier name should be a mnemonic name and can be of any length.
- Identifier names are case sensitive for example my name, and Name is not the same.

Arithmetic Operators:
Used to perform mathematical operations on operands.

Operator	Meaning	Example
+	Add two operands or unary plus	x + y +2
-	Subtract right operand from the left or unary minus	x - y -2
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y (remainder of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y (x to the power y)

Relational/Comparison Operators:
Relationship between two operands. Result is Boolean either True or False.

Comparision operators in Python

Operator	Meaning	Example
>	Greater that - True if left operand is greater than the right	x > y
<	Less that - True if left operand is less than the right	x < y
==	Equal to - True if both operands are equal	x == y
!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to - True if left operand is greater than or equal to the right	x >= y
<=	Less than or equal to - True if left operand is less than or equal to the right	x <= y

Logical Operators:
Result is Boolean either True or False.

Logical operators in Python

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Bitwise Operators:
Act on operands as if they are a string of binary values.
Let *x* = 10 (`0000 1010` in binary) and *y* = 4 (`0000 0100` in binary)

Bitwise operators in Python

Operator	Meaning	Example
&	Bitwise AND	x& y = 0 (`0000 0000`)
|	Bitwise OR	x | y = 14 (`0000 1110`)
~	Bitwise NOT	~x = -11 (`1111 0101`)
^	Bitwise XOR	x ^ y = 14 (`0000 1110`)
>>	Bitwise right shift	x>> 2 = 2 (`0000 0010`)
<<	Bitwise left shift	x<< 2 = 40 (`0010 1000`)

Assignment Operators:
Assignment operators are used in Python to create and assign values to variables.
`a = 5` is a simple assignment operator that creates and assigns the value 5 on the right to the variable *a* on the left.
There are various compound operators in Python like `a += 5` that adds to the variable and later assigns the same. It is equivalent to `a = a + 5`.

Operator	Example	Equivatent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

Python follows PEMDAS.
P = Parenthesis has higher priority.
E = Exponent (Power of)
M = Multiplication
D = Division
A = Addition
S = Subtraction has lowest priority

Identity Operators:

'is' and 'is not' are two identity operators to identify if two values are located in same part of the memory. Two variables that are equal does not imply they are identical.

Identity operators in Python

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

Example #4: Identity operators in Python

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

# Output: False
print(x1 is not y1)

# Output: True
print(x2 is y2)

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

Membership Operators:

'in' and 'not in' are two membership operators which are used to validate the membership of a value in a sequence such as strings, lists, tuples etc.

In a dictionary we can only test for presence of key, not the value.

Operator	Meaning	Example
In	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Example #5: Membership operators in Python

```
x = 'Hello world'
y = {1:'a',2:'b'}

# Output: True
print('H' in x)

# Output: True
print('hello' not in x)

# Output: True
print(1 in y)

# Output: False
print('a' in y)
```

Here, `'H'` is in *x* but `'hello'` is not present in *x* (remember, Python is case sensitive). Similary, `1` is key and `'a'` is the value in dictionary *y*. Hence, `'a' in y` returns `False`.

Python Modules:

- It is a python file containing python definitions and statements.
- A module can define functions, classes, variables.
- It allows you to logically organize your code.
- Grouping related code into a module makes the code easier to understand and use.
- To create a module, just save the file with **.py** as extension.
- By using the **import** statement, we can get the module and use it in another code.
- When the interpreter encounters an import statement, it imports the module if it is in the search path. A search path is a list of directories that the interpreter searches before importing a module.
    1. import : This is direct importing of the module without any changes with all the functions, statements to use.

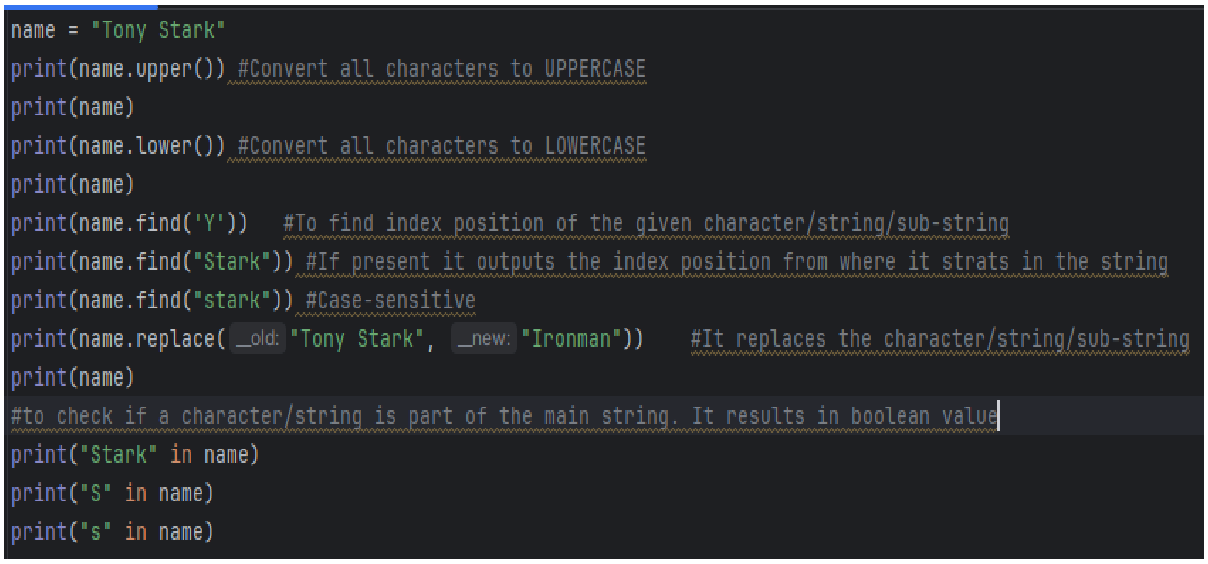
       import <module_name**>**
    2. import as : Using this we can create an alias for the module which has a bigger name.

       import <module_name**> as <**alias**>**
    3. from import : Using this we only import the specific function that we need from a module.

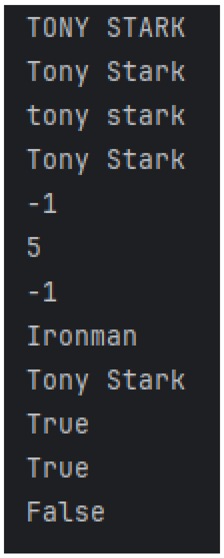
       from <module_name**> import <**function**>**

Python Strings :

- String represents an ordered sequence of characters.
- Strings can be created by enclosing characters inside a single quote or double quotes.
- To store multiple lines in a string variable use triple quotes. (‘’’ ‘’’) or use “**\n**”
- String Methods : There are so many methods built-in. Some of them are shown below.



```
name = "Tony Stark"
print(name.upper()) #Convert all characters to UPPERCASE
print(name)
print(name.lower()) #Convert all characters to LOWERCASE
print(name)
print(name.find('Y'))   #To find index position of the given character/string/sub-string
print(name.find("Stark")) #If present it outputs the index position from where it strats in the string
print(name.find("stark")) #Case-sensitive
print(name.replace( _old: "Tony Stark",  _new: "Ironman"))    #It replaces the character/string/sub-string
print(name)
#to check if a character/string is part of the main string. It results in boolean value
print("Stark" in name)
print("S" in name)
print("s" in name)
```



```
TONY STARK
Tony Stark
tony stark
Tony Stark
-1
5
-1
Ironman
Tony Stark
True
True
False
```

- strip()..........removes the spaces at the start and end of the string

String Validators :

- isalpha() : Checks if all character in string are alphabets(a-z, A-Z)
- isalnum() : Checks if all character in string are alphanumerics(a-z, A-Z, 0-9)
- isdigit() : Checks if all character in string are digits(0-9)
- islower() : Checks whether the characters are lowercase(a-z)
- isupper() : Checks whether the characters are uppercase(A-Z)

Python Lists :

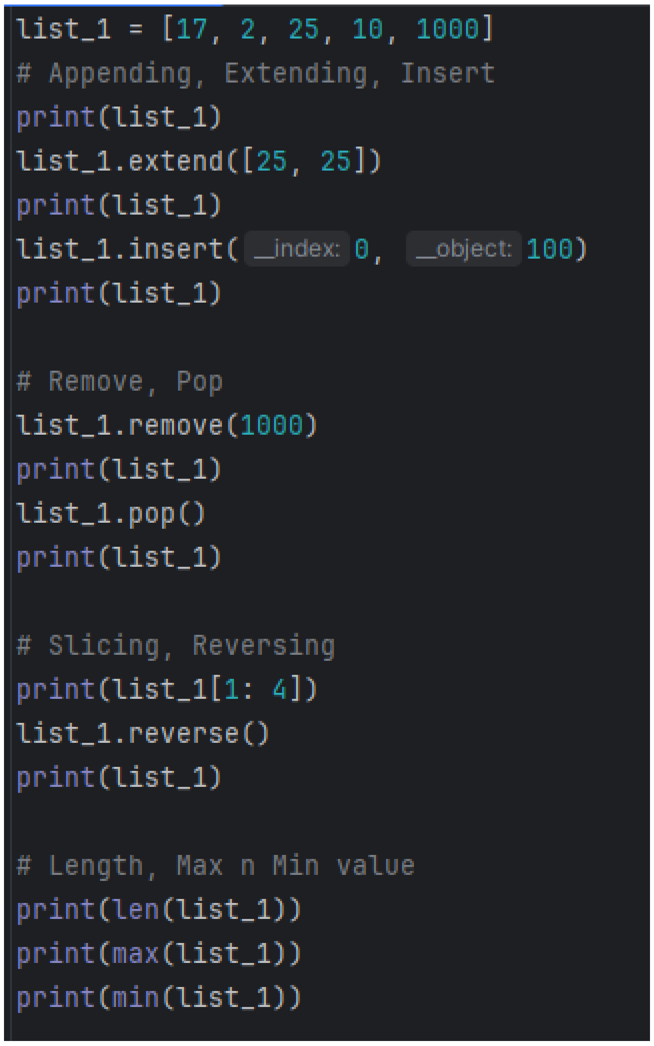
Python offers a set of compound data types referred to as sequences. A list is an ordered sequence of values/elements/items of same or different type. List is mutable.

A list is created by placing all the items (elements) inside a square bracket [ ], separated by commas. It can have any number of items and they may be of different types (integer, float, string etc.).

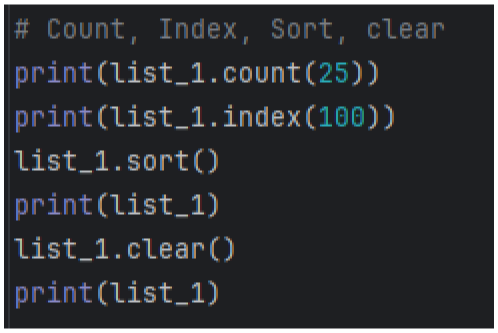
```
# empty list - list with no elements
my_list = []

# list of integers
my_list = [1, 2, 3]

# list with mixed datatypes
my_list = [1, "Hello", 3.4]
```

```
my_list = ['p','r','o','b','e']
# Output: p
print(my_list[0])

# Output: o
print(my_list[2])

# Output: e
print(my_list[4])
# Error! Only integer can be used for indexing
# my_list[4.0]

my_list = ['p','r','o','b','e']
# Output: e
print(my_list[-1]) #index = - points to the last element in the list

# Output: p
print(my_list[-5])
```

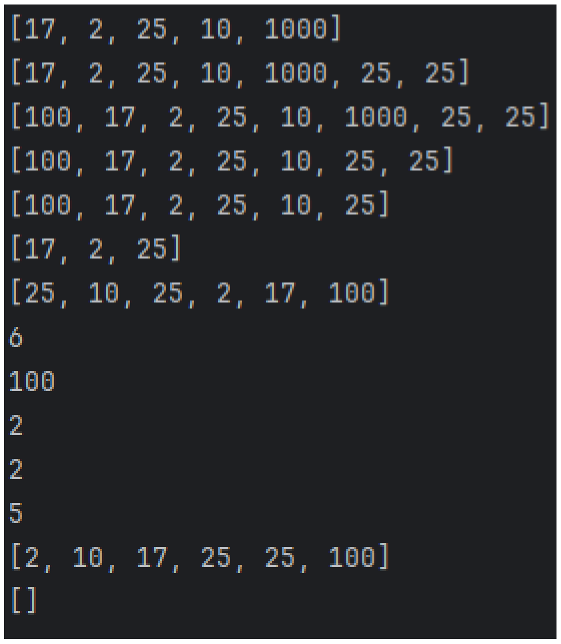
- List Operations :
    - append() …….. to add **single** value at the end of the list
    - extend([ ]) ………. to add more than one value at the end of the list
    - insert() ……… to add a value at specified index position
    - remove() ……. to remove the specified value from the list
    - pop() …….. to remove the value present at the end of the list
    - slice() …….. to get the exact values from the range specified
    - reverse() …….. to reverse the entire list
    - len(), min(), max() ……. to get the length, maximum and minimum value of the list
    - count() ……. to count the number of copies the value is stored in the list
    - index() …….. to get the index of the specified value
    - sort() ……. to arrange the values in ascending order
    - clear() ……. to clear the entire list



```
list_1 = [17, 2, 25, 10, 1000]
# Appending, Extending, Insert
print(list_1)
list_1.extend([25, 25])
print(list_1)
list_1.insert( __index: 0,  __object: 100)
print(list_1)

# Remove, Pop
list_1.remove(1000)
print(list_1)
list_1.pop()
print(list_1)

# Slicing, Reversing
print(list_1[1: 4])
list_1.reverse()
print(list_1)

# Length, Max n Min value
print(len(list_1))
print(max(list_1))
print(min(list_1))
```



```
# Count, Index, Sort, clear
print(list_1.count(25))
print(list_1.index(100))
list_1.sort()
print(list_1)
list_1.clear()
print(list_1)
```



```
[17, 2, 25, 10, 1000]
[17, 2, 25, 10, 1000, 25, 25]
[100, 17, 2, 25, 10, 1000, 25, 25]
[100, 17, 2, 25, 10, 25, 25]
[100, 17, 2, 25, 10, 25]
[17, 2, 25]
[25, 10, 25, 2, 17, 100]
6
100
2
2
5
[2, 10, 17, 25, 25, 100]
[]
```

Python Tuple:

Tuple is an ordered sequence of immutable python elements of same of different types. Unlike lists, tuple uses parenthesis ( ).

A tuple is created by placing all the items (elements) inside a parentheses (), separated by comma. A tuple can have any number of items and they may be of different types (integer, float, list, string etc.).

```
# empty tuple
# Output: ()
my_tuple = ()
print(my_tuple)

# tuple having integers and Output: (1, 2, 3)
my_tuple = (1, 2, 3)
print(my_tuple)

# tuple with mixed datatypes
# Output: (1, "Py", 3.4)
my_tuple = (1, "Py", 3.4)
print(my_tuple)

#creating a tuple with a single element
>>>t=('a',)
('a',)
```

```
# only parentheses is not enough
# Output: <class 'str'>
my_tuple = ("hello")
print(type(my_tuple))

# need a comma at the end
# Output: <class 'tuple'>
my_tuple = ("hello",)
print(type(my_tuple))
```

Unlike lists, tuple is immutable ie cannot undergo changes. If we try to change it provides a TypeError.

We can use + operator to combine two tuples. This is also called **concatenation**.
We can also **repeat** the elements in a tuple for a given number of times using the * operator.
Both + and * operations result into a new tuple.

```
# Concatenation
# Output: (1, 2, 3, 4, 5, 6)
print((1, 2, 3) + (4, 5, 6))

# Repeat
# Output: ('Repeat', 'Repeat', 'Repeat')
print(("Repeat",) * 3)
```

- Tuple Methods
    - count().......... returns the total count of the specified element inside the tuple
    - index().......... returns the index value of the element inside the tuple. For the element which is repeated multiple times, it returns the 1st appearance's index value

Python Sets:

Sets are unordered collection of unique elements/items of same or different types. Sets are mutable. Order of elements in a set is undefined. It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have a mutable element, like list, set or dictionary, as its element.

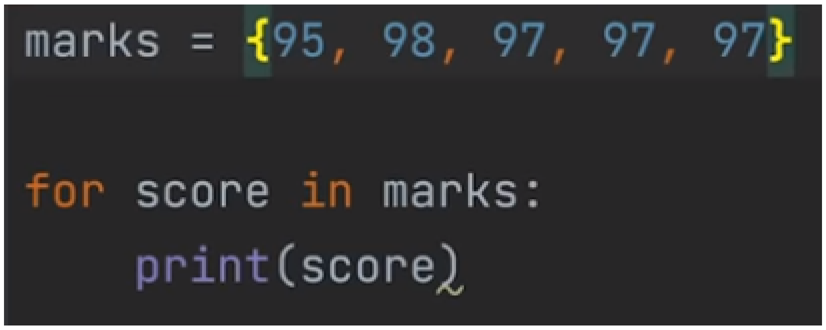
A set is created by placing all the items (elements) inside curly braces {}, separated by comma or by using the built-in function `set()`.

```
# set of integers
# output : {2,1,3}
my_set = {1, 2, 3}
print(my_set)
```

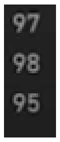
Since a set is unordered, each time a set is printed, the elements are not in an order.

```
# set do not have duplicates ie duplicate items will be replaced with a single element
# Output: {1, 2, 3, 4}
my_set = {1,2,3,4,3,2}
print(my_set)
>>>A=set('querty') # converting a string in to a set
>>>print(A)
{'t','q','u','r','e','y'}
```

For accessing the elements in sets



```
marks = {95, 98, 97, 97, 97}

for score in marks:
    print(score)
```



```
97
98
95
```

Operations on Sets:

We can know the length or the number of elements in a set by applying len( ) function. To add elements on to a set we can use add( ) function.

```
#Output = 5
s = {1,3,4,5,6}
print(len(s))

# TypeError: 'set' object does not support indexing
s[0]

# add an element. Output: {1, 2, 3,6,5,4}
s.add(2)
print(s)
```

Two methods/Functions are available to remove elements from a set: they are discard( ) and remove( ).

discard( ): Just deletes element from a set if present, if not present it does nothing.
remove( ): Removes element from a set if present, if not raises an exception ‘KeyError’.

```
st = {1, 3, 4, 5, 6}

# discard an element
# Output: {1, 3, 5, 6}
st.discard(4)
print(st)

# remove an element
# Output: {1, 3, 5}
st.remove(6)
print(st)

# discard an element
# not present in st
# Output: {1, 3, 5}
st.discard(2)
print(st)
```

Python Dictionary:

Dictionary is an unordered mutable collection of same or different types of elements. Each element in a dictionary consists of a key:value pair. Elements of a dictionary are not accessed using an index value but using a key associated with each value. Values can be of any data type, key should be of immutable data type(number, string or tuple) and must be unique.

Dictionary can be created by placing elements within {} braces and separated by a comma operator. Dictionary can also be created using dict( ) function.

```
my_dt = {}  #Empty dictionary

# dictionary with integer keys
my_dt1 = {1: 'apple', 2: 'ball'}
```

```
# dictionary with mixed keys
my_dt2 = {'name': 'John', 1: [2, 4, 3]}

# using dict()
my_dt3 = dict({1:'apple', 2:'ball'})
```

Dictionary are mutable. We can add new items or change the value of existing items using assignment operator.
If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.

```
my_dt = {'name':'Py', 'age': 26}

# update value
my_dt['age'] = 27

#Output: {'age': 27, 'name': 'Py'}
print(my_dt)
```

```
# add item
my_dt['address'] = 'Downtown'

# Output: {'address': 'Downtown', 'age': 27, 'name': 'Py'}

  print(my_dt)
```

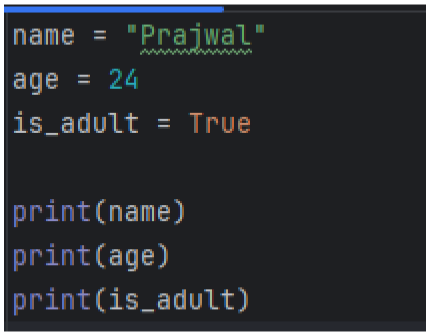
We can remove a particular item in a dictionary by using the method `pop()`. This method removes as item with the provided key and returns the value.
The method, `popitem()` can be used to remove and return an arbitrary item (key, value) form the dictionary. All the items can be removed at once using the `clear()` method.
We can also use the `del` keyword to remove individual items or the entire dictionary itself.

```
# create a dictionary
squares = {1:1, 2:4, 3:9, 4:16, 5:25}

# remove a particular item. Output: 16
print(squares.pop(4))

# Output: {1: 1, 2: 4, 3: 9, 5: 25}
print(squares)

# remove an arbitrary item. Output: (1, 1)
print(squares.popitem())

# Output: {2: 4, 3: 9, 5: 25}
print(squares)
```

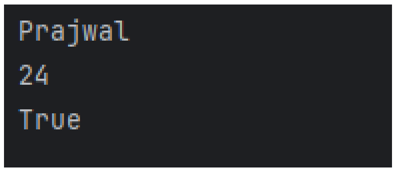
```
# delete a particular item
del squares[5]

# Output: {2: 4, 3: 9}
print(squares)

# remove all items
squares.clear()

# Output: {}
print(squares)

# delete the dictionary itself
del squares

# Throws Error
print(squares)
```

Variables :

- Used as a Memory location.
- No need to specify the variable type. It automatically get the type based on the value it is assigned with.
- Name of a variable is an identifier.
- It can contain letters, numbers and underscore (_). But it must start with either letter or an underscore.
- Values may be a string, character or an integer.
- Types:
  I. String
  II. Number : Integer and Floating
  III. Boolean : **True** or **False**



```
name = "Prajwal"
age = 24
is_adult = True

print(name)
print(age)
print(is_adult)
```



```
Prajwal
24
True
```

<u>Scope of Variables :</u>

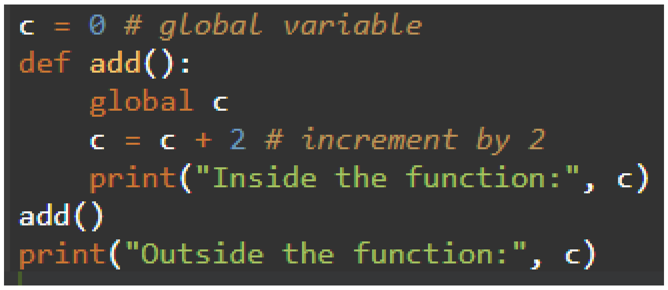
- All variables are not allowed to use in every portion of the code.
- This depends on where they are declared.
- Scope determines the portion of code where that variable can be used.
- Two types :
  - Global Variables :
    - Variables which are not declared inside any function and are part of main code then they are global variables which can be used anywhere in the entire program.
  - Local Variables :
    - Variables which are declared inside function are called local variables which can be used only inside the function where it is declared.

Global Keyword:

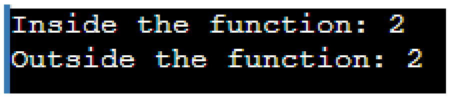
In Python, global keyword allows you to modify the variable outside of the current scope. It is used to create a global variable and make changes to the variable in a local context.

We use global keyword to read and write a global variable inside a function.

With Global keyword:

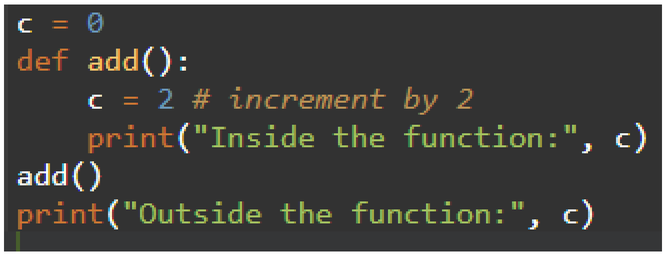


```python
c = 0 # global variable
def add():
    global c
    c = c + 2 # increment by 2
    print("Inside the function:", c)
add()
print("Outside the function:", c)
```

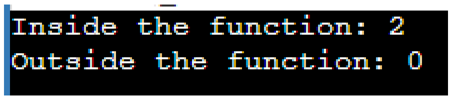


```
Inside the function: 2
Outside the function: 2
```

Without global keyword:



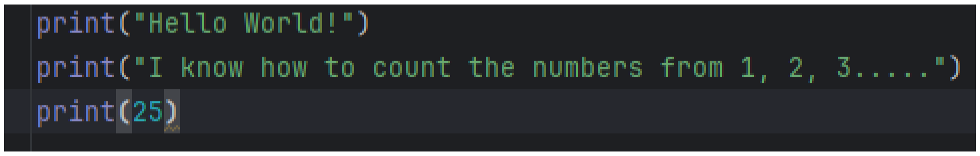
```python
c = 0
def add():
    c = 2 # increment by 2
    print("Inside the function:", c)
add()
print("Outside the function:", c)
```



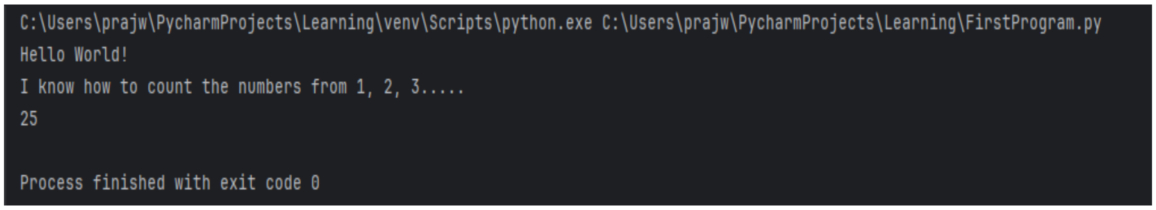
```
Inside the function: 2
Outside the function: 0
```

In-Built Functions :

1. **print()**
    - Func which returns the value present inside the parentheses.
    - Inside the parentheses, for strings and characters we should use **"<string/character>"** or **'<string/character>'**.
    - For integers no need of quotes. Can directly give the value inside the parentheses.

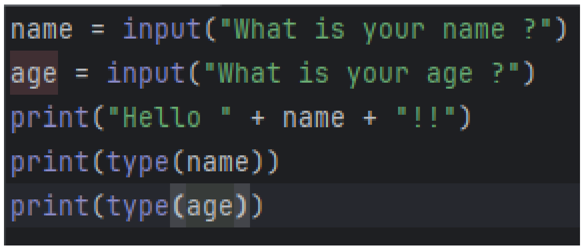


```
print("Hello World!")
print("I know how to count the numbers from 1, 2, 3.....")
print(25)
```

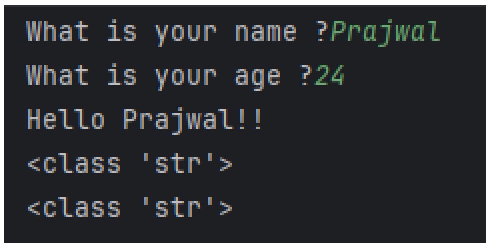


```
C:\Users\prajw\PycharmProjects\Learning\venv\Scripts\python.exe C:\Users\prajw\PycharmProjects\Learning\FirstProgram.py
Hello World!
I know how to count the numbers from 1, 2, 3.....
25

Process finished with exit code 0
```

2. **input()**
    - Function used to take input from the user.
    - Whatever inside the parentheses will be displayed. It should be relevant to what kind of input the user needs to give.
    - Whatever be the input type the user gives, but **it'll be stored as a string**.



```
name = input("What is your name ?")
age = input("What is your age ?")
print("Hello " + name + "!!")
print(type(name))
print(type(age))
```



```
What is your name ?Prajwal
What is your age ?24
Hello Prajwal!!
<class 'str'>
<class 'str'>
```

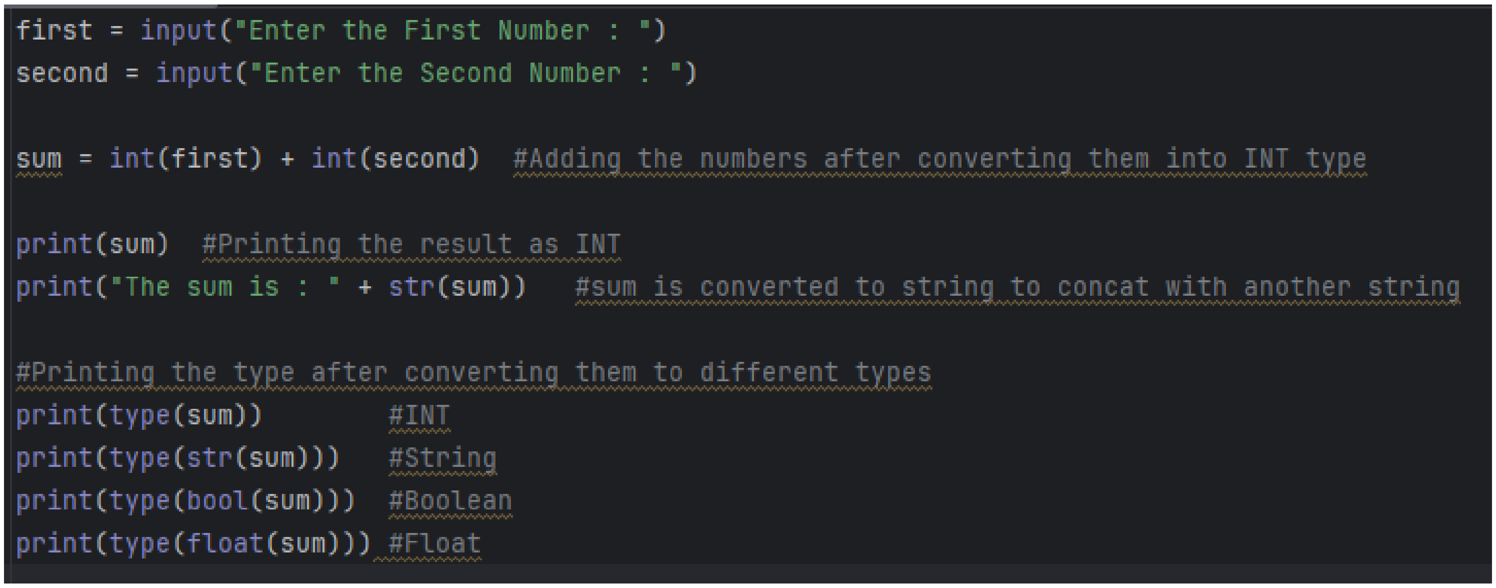
➢ **+** operator can be used as a **Concatenation operator.**

3. **split()**
    - Used to split the input into list for every space and can be accessed by indexing.

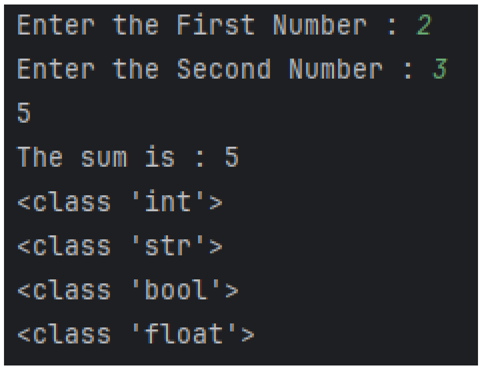
4. **join()**
    - Used to join the elements

5. **return()**
   - Used to return any value from the function to the point where the function is called.
6. **divmod(<**x**>,<**y**>)**
   - It returns both quotient and remainder in tuple.
7. **round(<**number**>,<**digits_to_round_of**>)**
   - Used to round any floating values.
8. **del**
   - Used to delete anything which comes inside its scope.
9. **pow(<**x**>,<**y**>,<**z**>)**
   - x raised to y and divide by z returns the remainder.
   - Equivalent to **(x**y)%z,** which gives you the remainder.

10. **Type Conversions : [** Explicit Conversion **]**
    a. **int()**
    b. **float()**
    c. **str()**
    d. **bool()**



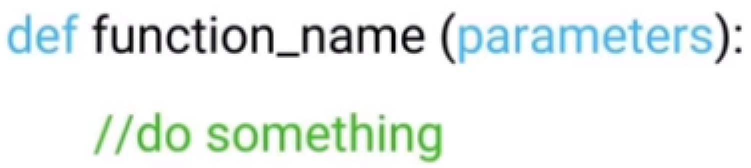
```
first = input("Enter the First Number : ")
second = input("Enter the Second Number : ")

sum = int(first) + int(second)  #Adding the numbers after converting them into INT type

print(sum)  #Printing the result as INT
print("The sum is : " + str(sum))   #sum is converted to string to concat with another string

#Printing the type after converting them to different types
print(type(sum))        #INT
print(type(str(sum)))   #String
print(type(bool(sum)))  #Boolean
print(type(float(sum))) #Float
```



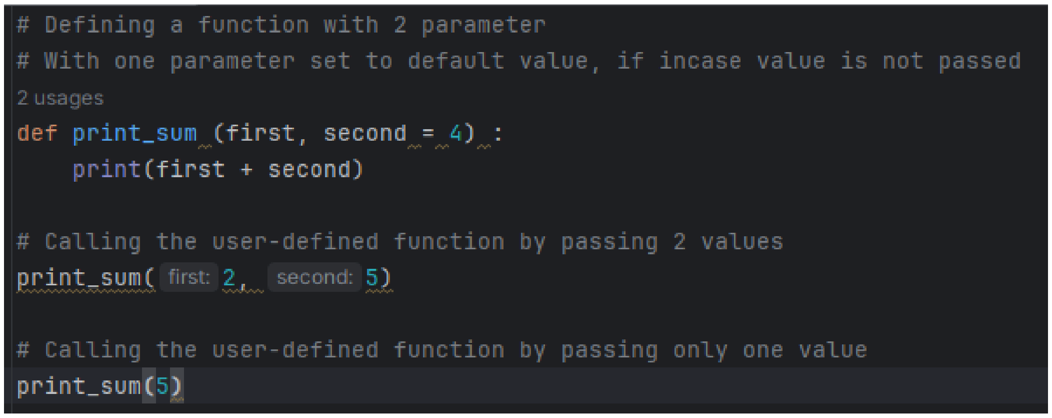
```
Enter the First Number : 2
Enter the Second Number : 3
5
The sum is : 5
<class 'int'>
<class 'str'>
<class 'bool'>
<class 'float'>
```

User-Defined Functions :

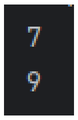
- These are the functions which will not be a part of in-built functions.
- They will be created by the user as per their requirements and tasks.
- There can be a return**()** function at the end of the function body, which primarily returns the value which is specified to the point where the function is called and secondly it exits from the function.
- **return()** is optional.



```
def function_name (parameters):

    //do something
```



```
# Defining a function with 2 parameter
# With one parameter set to default value, if incase value is not passed
2 usages
def print_sum (first, second = 4) :
    print(first + second)

# Calling the user-defined function by passing 2 values
print_sum( first: 2,  second: 5)

# Calling the user-defined function by passing only one value
print_sum(5)
```

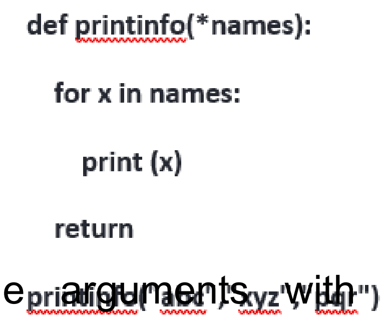


```
7
9
```

- Function Arguments:
    - Required Arguments :
        - Required arguments are the arguments passed to a function in correct **positional order**.
        - The number of arguments in the function call should match exactly with the function definition.
    - Keyword Arguments :
        - Keyword arguments are related to function calls.
        - When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

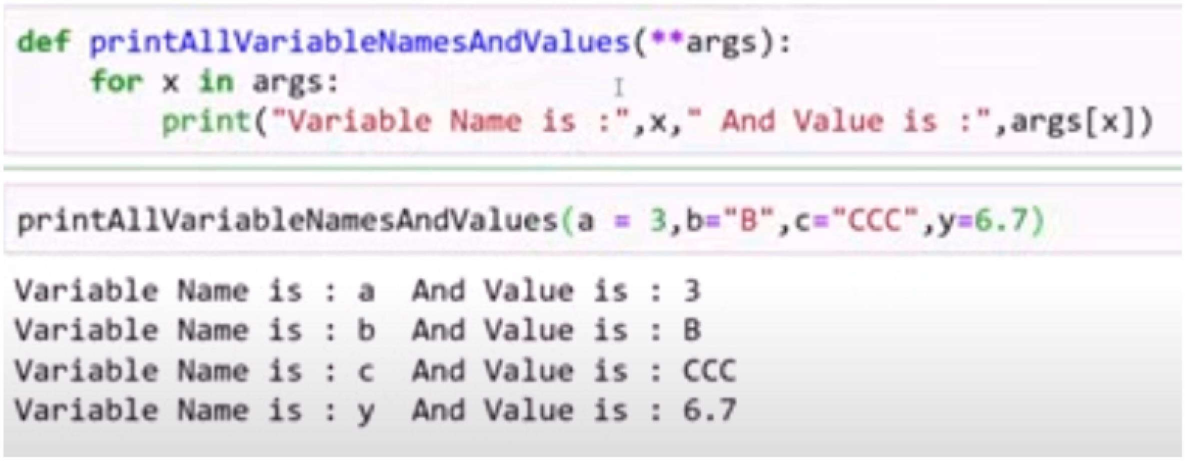
```
def printme(x,y):

    print(x)

    print(y)

    return

printme(x=2,y=3)

printme(y=3,x=2)
```

- Default Arguments :
    - Default arguments is an argument that assumes a default value when the value is not passed while calling the function.
- Variable-length Arguments :
    - Sometimes, when we don’t know in advance the number of arguments, then python allows us to use arbitrary numbers of arguments.
    - In the function definition we **asterisk(*) before the parameter name**.



```
def printinfo(*names):
  for x in names:
    print (x)
  return
printinfo("abc","xyz","pqr")
```

    - For multiple arguments with multiple variables use **double asterisk(**)** before the parameter inside the function definition. This will be stored in a dictionary manner (Not the actual dictionary type of python). For parsing through them use the variable ref as the index position.



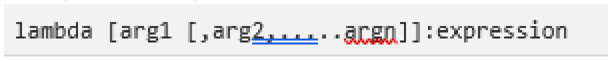
```
def printAllVariableNamesAndValues(**args):
    for x in args:
        print("Variable Name is :",x," And Value is :",args[x])
```

```
printAllVariableNamesAndValues(a = 3,b="B",c="CCC",y=6.7)
```

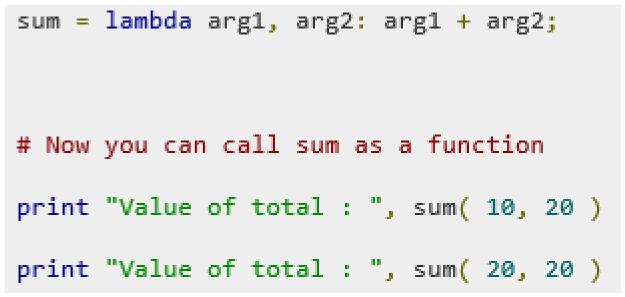
```
Variable Name is : a  And Value is : 3
Variable Name is : b  And Value is : B
Variable Name is : c  And Value is : CCC
Variable Name is : y  And Value is : 6.7
```

Lambda Functions :

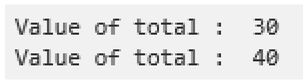
- Their functions are called anonymous because they are not declared with proper manner by using **def** keyword.
- Using the **lambda** keyword, you can create small anonymous functions.



```
lambda [arg1 [,arg2,.....argn]]:expression
```



```
sum = lambda arg1, arg2: arg1 + arg2;


# Now you can call sum as a function

print "Value of total : ", sum( 10, 20 )

print "Value of total : ", sum( 20, 20 )
```



```
Value of total :  30
Value of total :  40
```

Statements :

- 2 types
    - Conditional / Branching statements.

      Conditional statements are required when we want to execute a set of statements only if the condition is satisfied.

        i. if
        ii. if-else
        iii. if-elif-else
        iv. Nested if

    - Looping / Iterative statements.

      Statements used for repetitive execution.

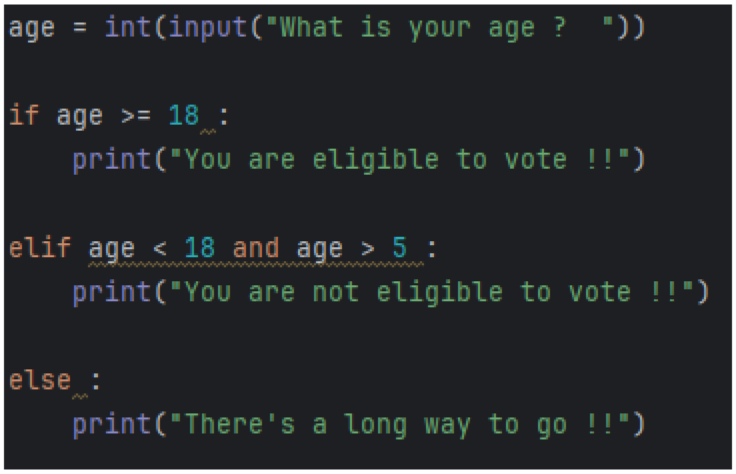
        i. for
        ii. range
        iii. while
        iv. break and continue

➢ Conditional / Branching statements :

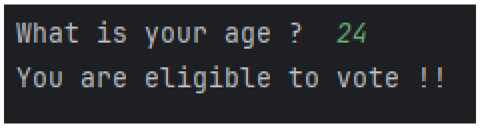
```
if conditional_expression:
    statement(s)
```

```
if conditional_expression:
    Body of if
else:
    Body of else
```

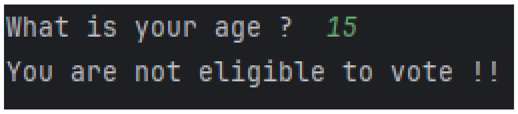
```
if test expression:
    Body of if
elif test expression:
    Body of elif
else:
    Body of else
```



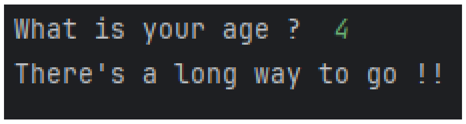
```
age = int(input("What is your age ?  "))

if age >= 18 :
    print("You are eligible to vote !!")

elif age < 18 and age > 5 :
    print("You are not eligible to vote !!")

else :
    print("There's a long way to go !!")
```



```
What is your age ?  24
You are eligible to vote !!
```



```
What is your age ?  15
You are not eligible to vote !!
```



```
What is your age ?  4
There's a long way to go !!
```

➢ Looping / Iterative statements :

- range()

  We can generate a sequence of numbers using `range()` function.

  We can also define the start, stop and step size as `range(start,stop,step size)`. step size defaults to 1 if not provided.

  This function does not store all the values in memory.

  To force this function to output all the items, we can use the function `list()`.

```
# Output: range(0, 10)
print(range(10))

# Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(list(range(10)))
```

- while()

  While loop is called an indefinite loop.

  The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.
  We generally use this loop when we don't know beforehand, the number of times to iterate.

  Syntax of while Loop in Python

```
while conditional_expression:
    Body of while
```

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

- for()

  For statement is a definitive looping statement. The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

Syntax:
```
for index in sequence:
        Body of for
```
Here, `index` is the variable that takes the value of the item inside the sequence on each iteration.

- break and continue

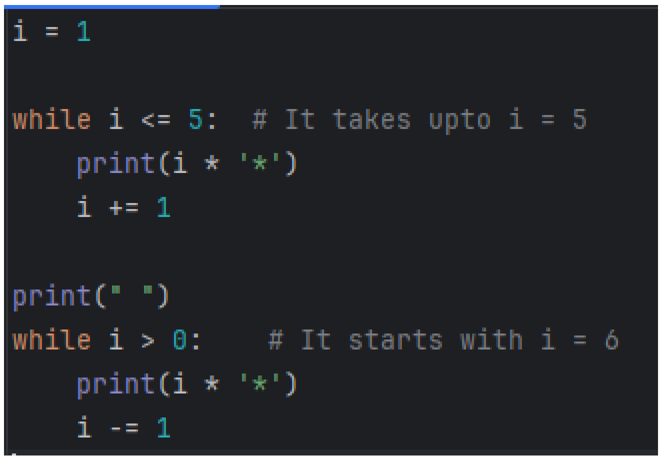
Break and continue can be used to alter the flow of a looping statement. Break can also be used along with Conditional/Branching statements.

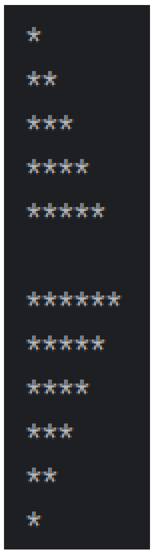
The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

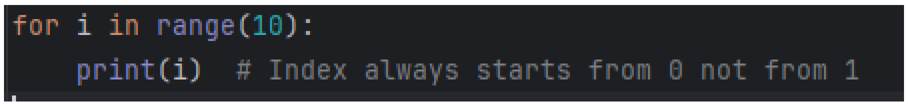
Syntax of break:
```
break
```

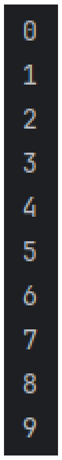
The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

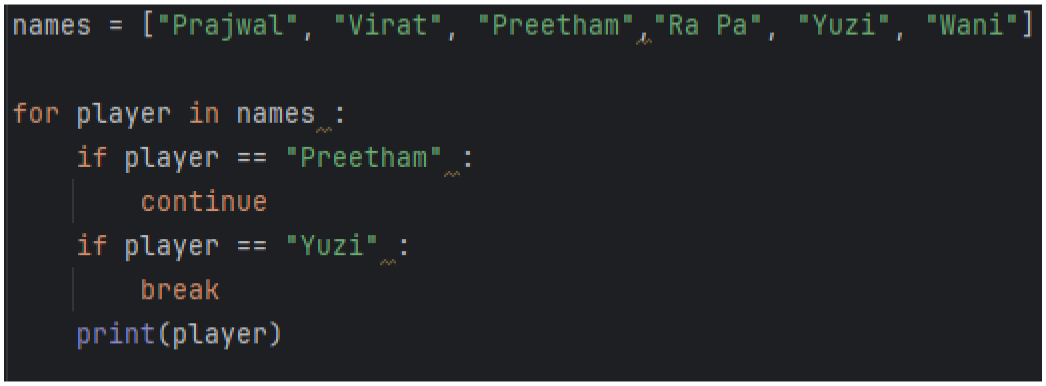
Syntax of Continue
```
Continue
```

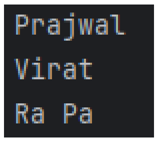

```
i = 1

while i <= 5:  # It takes upto i = 5
    print(i * '*')
    i += 1

print(" ")
while i > 0:    # It starts with i = 6
    print(i * '*')
    i -= 1
```


```
*
**
***
****
*****

******
*****
****
***
**
*
```

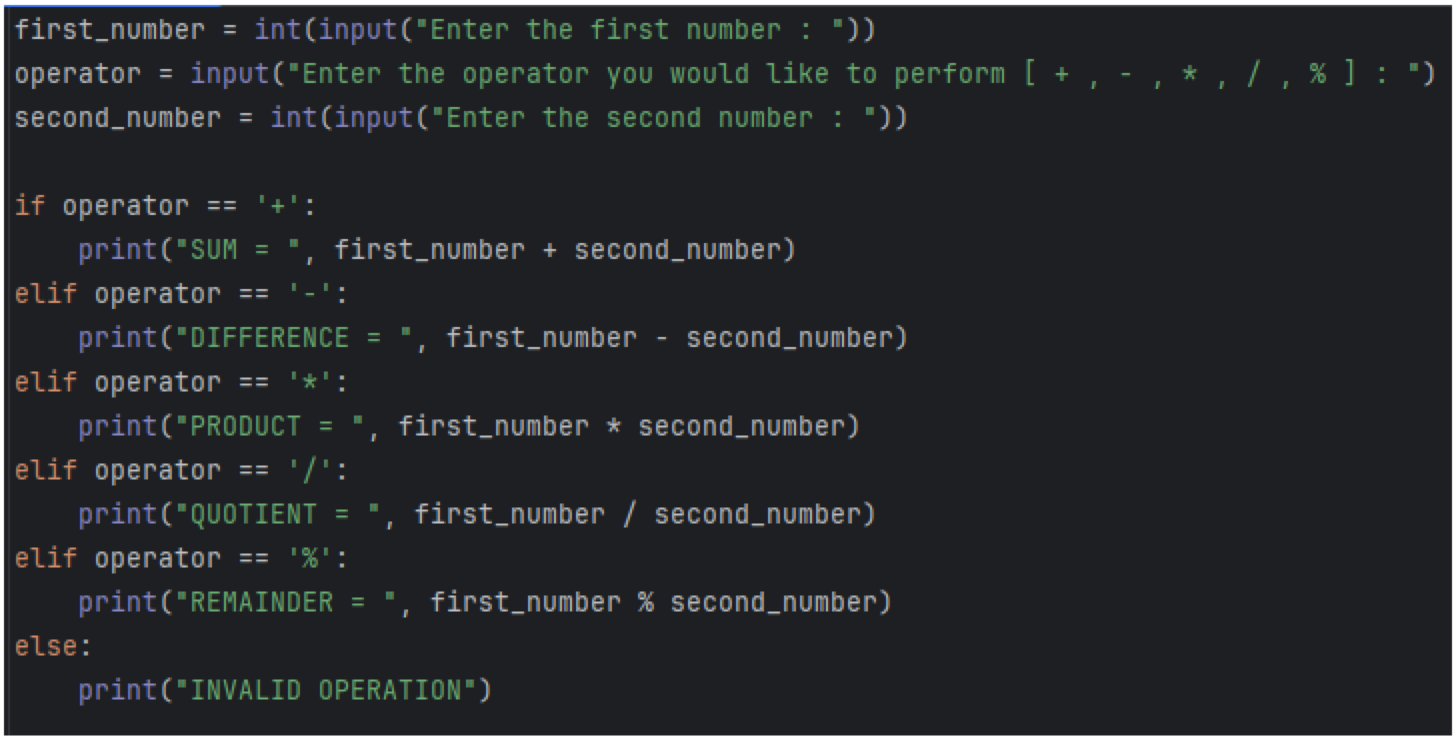

```
for i in range(10):
    print(i)  # Index always starts from 0 not from 1
```


```
0
1
2
3
4
5
6
7
8
9
```

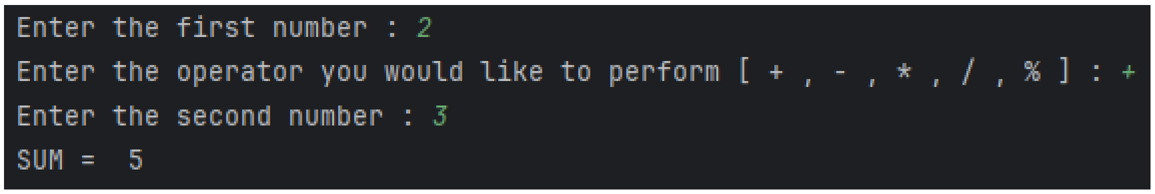

```python
names = ["Prajwal", "Virat", "Preetham","Ra Pa", "Yuzi", "Wani"]

for player in names :
    if player == "Preetham" :
        continue
    if player == "Yuzi" :
        break
    print(player)
```


```
Prajwal
Virat
Ra Pa
```

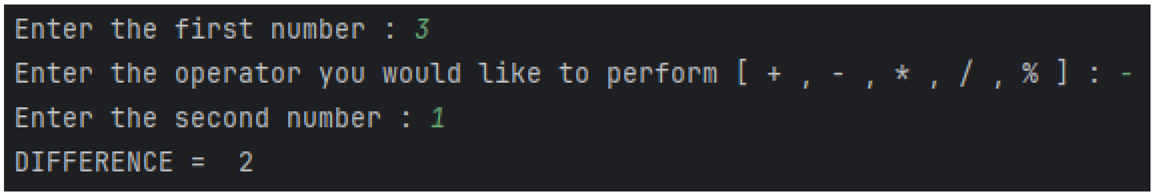
MINI CALCULATOR :



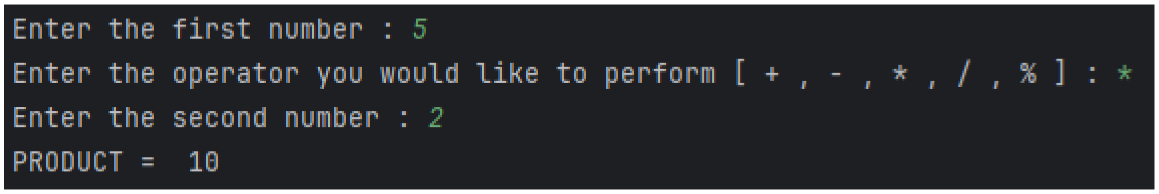
```python
first_number = int(input("Enter the first number : "))
operator = input("Enter the operator you would like to perform [ + , - , * , / , % ] : ")
second_number = int(input("Enter the second number : "))

if operator == '+':
    print("SUM = ", first_number + second_number)
elif operator == '-':
    print("DIFFERENCE = ", first_number - second_number)
elif operator == '*':
    print("PRODUCT = ", first_number * second_number)
elif operator == '/':
    print("QUOTIENT = ", first_number / second_number)
elif operator == '%':
    print("REMAINDER = ", first_number % second_number)
else:
    print("INVALID OPERATION")
```



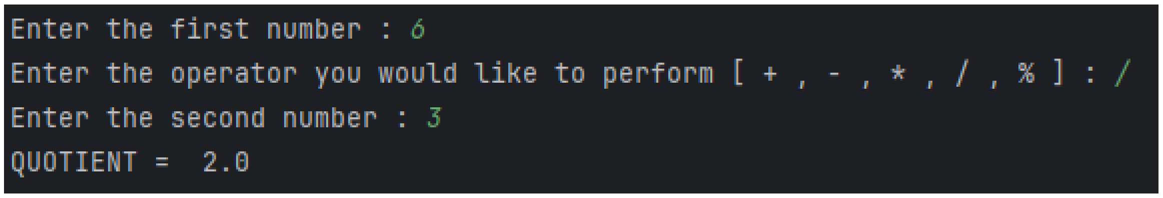
```
Enter the first number : 2
Enter the operator you would like to perform [ + , - , * , / , % ] : +
Enter the second number : 3
SUM =  5
```



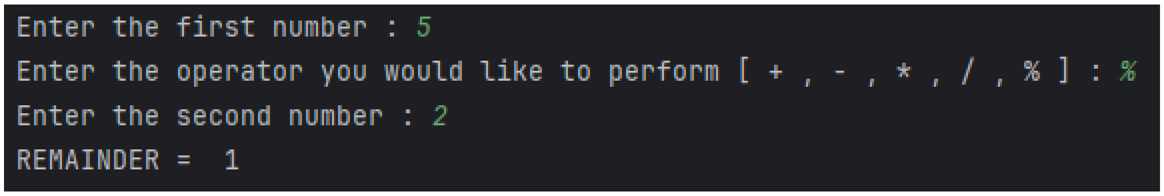
```
Enter the first number : 3
Enter the operator you would like to perform [ + , - , * , / , % ] : -
Enter the second number : 1
DIFFERENCE =  2
```



```
Enter the first number : 5
Enter the operator you would like to perform [ + , - , * , / , % ] : *
Enter the second number : 2
PRODUCT =  10
```



```
Enter the first number : 6
Enter the operator you would like to perform [ + , - , * , / , % ] : /
Enter the second number : 3
QUOTIENT =  2.0
```

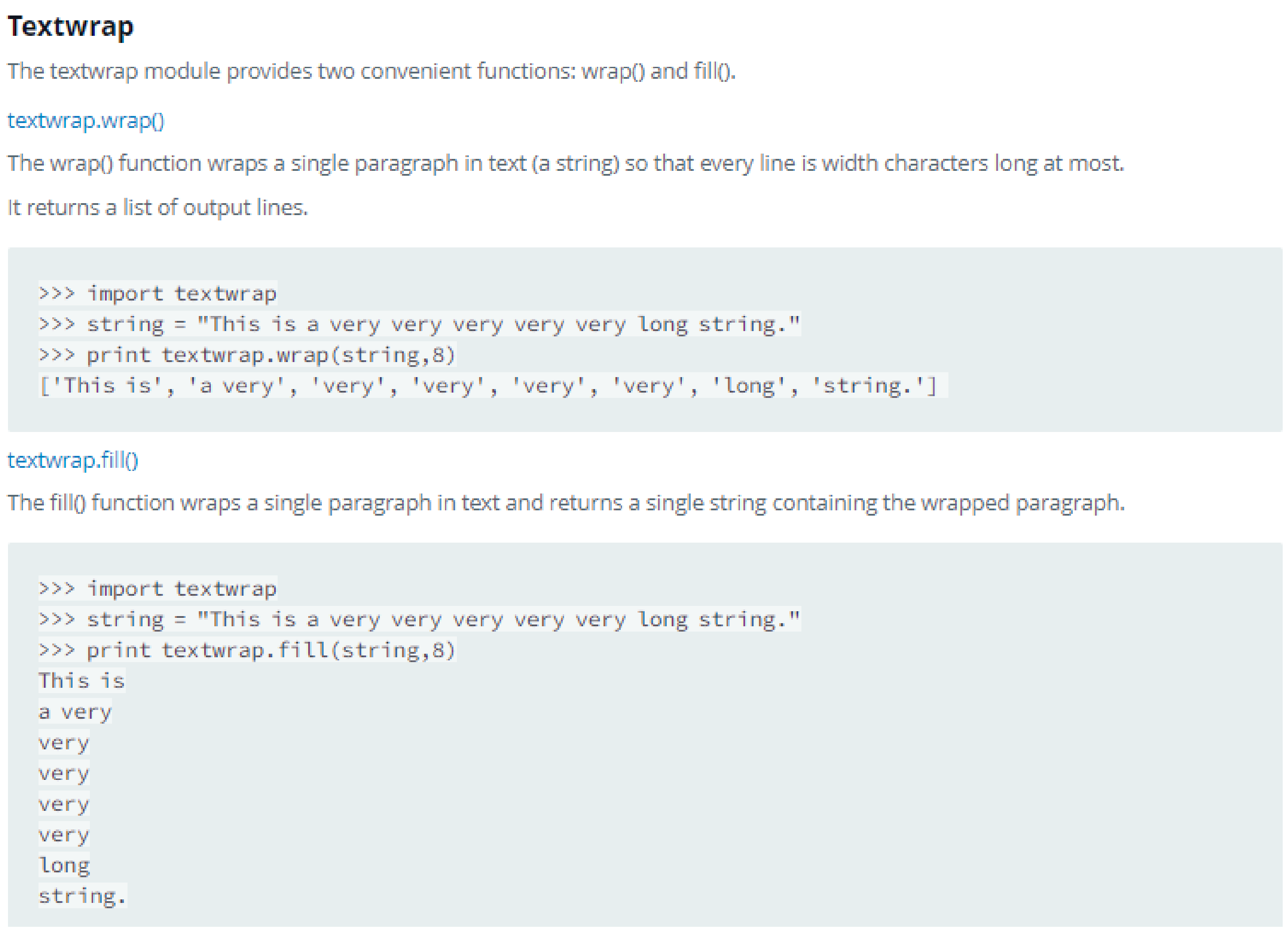


```
Enter the first number : 5
Enter the operator you would like to perform [ + , - , * , / , % ] : %
Enter the second number : 2
REMAINDER =  1
```

Leap Year Logic

- It’s a leap year if it is evenly divided by 4 and not evenly divided by 100
- It’s a leap year if it is evenly divided by both 400 and 100

Programming for placements, competitive : OOPS
AI, ML, Data Science : Different modules for them with file handling
Development side : Django framework



Textwrap

The textwrap module provides two convenient functions: wrap() and fill().

textwrap.wrap()

The wrap() function wraps a single paragraph in text (a string) so that every line is width characters long at most.

It returns a list of output lines.

```
>>> import textwrap
>>> string = "This is a very very very very very long string."
>>> print textwrap.wrap(string,8)
['This is', 'a very', 'very', 'very', 'very', 'very', 'long', 'string.']
```

textwrap.fill()

The fill() function wraps a single paragraph in text and returns a single string containing the wrapped paragraph.

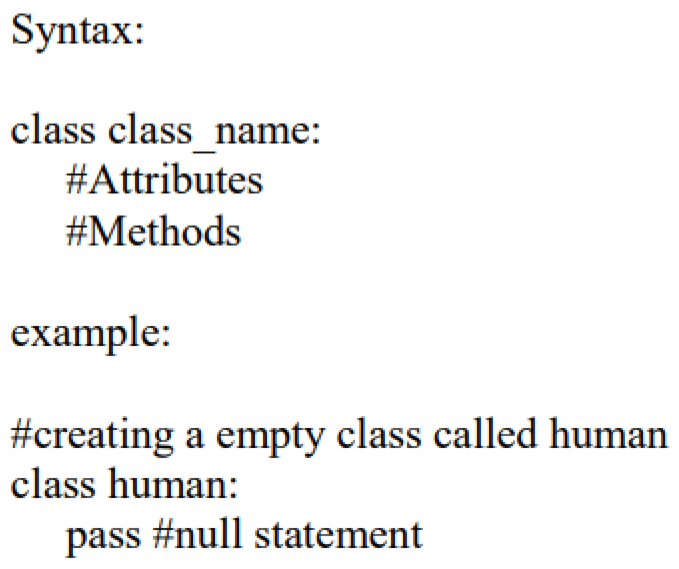
```
>>> import textwrap
>>> string = "This is a very very very very very long string."
>>> print textwrap.fill(string,8)
This is
a very
very
very
very
very
long
string.
```

OOPS

- ❖ Object Oriented Programming deals with programs which are oriented towards real world object having their own properties and behavior
- ❖ Real world objects could represent humans with properties such as name, age, weight, height, gender…… and behavior such as walking, talking, swimming…..
- ❖ Object has two characteristics:
    - ➢ Property [Attributes]
    - ➢ Behaviour [Methods]
- ❖ Features / Principles of OOPS :
    - ➢ **Encapsulation :** Binding both data(Attr) and operations(Methods) into a single closed unit called Encapsulation. This leads to **Data Abstraction [** Data Hiding **]**
    - ➢ **Inheritance :** Creating a new class [Derived] from an existing class [Parent/Base]. This is leads to reusability of the code.
    - ➢ **Polymorphism :** Concept of providing single entity with multiple functionality.

Class :

- ➢ Is a blueprint of an object or instance.
- ➢ Is used to create user defined datatypes, using which we can create real world objects/instances
- ➢ Contains both attributes and methods of an object/instance.
- ➢ Class definition only declared attributes and methods of an object does not associate any storage



```
Syntax:

class class_name:
    #Attributes
    #Methods

example:

#creating a empty class called human
class human:
    pass #null statement
```

Object / Instance of class :

- ➢ Can be created only after the creation of the class.
- ➢ Object of the class contains the characteristics of that class i.e., attributes and methods.
- ➢ Any number of objects/instances can be created.
- ➢ It can access both attributes and methods using dot (.) operator

```
Syntax:

obj_name = class_name( )

Ex:
#creating an instance called male of human class.
male = human( )
```

Methods :

- ➢ The behaviour of an object is called methods under class definition
- ➢ Class methods are defined under the scope of the class by using 'def' keyword.
- ➢ Methods are invoked/called normally just like a normal function.
- ➢ Methods are used to perform operations on an object/instance of a class

```
Syntax:

def method_name(optional_arguments):
    #optional instance attributes declaration
    # statements
```

The 'self' parameter:

1) Methods of a class must have a mandatory first argument/parameter called 'self' in the method definition even if the method does not take any argument. When the method is invoked, we do not provide value for the 'self' parameter, Python provides value to it.
2) Self parameter refers to the object/instance itself which accesses the method.

Attributes :

- ➢ The properties of an object is called an attribute under class definition.
- ➢ Attributes act as a container of information.
- ➢ Attributes are declared just like a normal variable and belong to the scope of the class.

```
Syntax:
attri_name = value
```

- ➢ Two types :
    - ○ Instance/Object Attributes :

Instance/Object attributes are attributes whose value are assigned inside a method(such as . Constructor) using a self parameter. Instance attributes can be accessed outside the class using a (.) dot operator through either an instance/object of a class or through the class name.

- Class Attributes :

  Class attributes are attributes assigned values within a class outside a method. Class attributes can be accessed outside the class using a (.) dot operator only through a class name.

Constructors :

- ➢ Special methods of class
- ➢ Used to create an object of a class
- ➢ Invoked/called when an object/instance of a class is created
- ➢ Used to initialize an objects attributes with initial values at the time of creation of an object

```
Syntax:
def _ _ init _ _(self):
        #statements
```

```
class human:
        “””Defining a class called human”””
        #Class attribute
        species = ‘mammals’
        #Constructor method
        def _ _init_ _(self, n, a, g):
                # Instance attributes
                self.name = n
                self.age = a
                self.gender = g
        def prnt(self):
                print(“Name is”, self.name,”\n”, “Age = “, self.age,”\n”,”gender = “, self.gender)


#defining an instance/object male of a class human. Constructor is invoked automatically
male = human(“Man”, “50”, ‘M’ )
#method of a class is invoked using the object of a class and dot (.) operator
male.prnt( )

Output:
Name is Man
Age = 50
gender = M
```

Destructors :

- ➢ Opposite to Constructors
- ➢ They are used to destroy an object when it goes out of scope.

- ➢ Destructors need not be defined manually as they are invoked automatically.
- ➢ In python, Objects of a class and its attributes can be deleted manually by using the keyword '**del**'.
- ➢ **'del'** statement can delete both objects as well as its attributes
- ➢ The automatic deletion/destruction of objects attributes is called **Garbage Collection**

Making attributes private to a class :

- ➢ Attributes can be accessed outside the class using the dot operator through object/class name.
- ➢ In order to cut down the access to attributes outside the class, we make attributes private which are accessible only inside the class and not anywhere.

Syntax:

```
_ _attribute_name = value
```

Example:

```
class MyClass:
        _ _hiddenvar = 0

        def add(self, increment):
                self._ _hiddervar + = increment
                print(self._ _hiddervar)

obj = MyClass( )-

obj.add(2)  # output = 2
obj.add(5)  # output = 7

print(obj._ _hiddervar)
#The above statement will result in an exception called 'AttributeError' as private members are not
# accessible
```

Built - in Attributes :

_ _dict_ _ : Provides dictionary holding the names present in a class definition.

_ _doc_ _: Provides class documentation (docstring) if present. If docstring is not present provides None value.

_ _name_ _: Provides the class name

_ _module_ _: Provides the name of the program in which the class is defined. In interactive mode it provides the name _ _main_ _

_ _ bases_ _ : Provides a tuple containing base class names in order of occurrence.

Inheritance

➢ Means creating a **new class** [**Derived / Child**] from an already existing **old class** [**Base / Parent**].
➢ Gives an advantage of reusability of an existing code.
➢ Inheritance provides 'is a' relationship

```
Syntax:

class BaseClassNm:
        # Base Class Attributes
        # Base Class Methods

class DerivedClassNm (BaseClassNm):
        #Derived Class Attributes
        #Derived Class Methods
```

➢ Derived Class inherits characteristics/ features of base class to its own characteristics/features

```
Example 1:
class Person:
        """ Base Class"""
        def _ _init_ _(self, fn, ln):
                self.firstname = fn
                self.lastname = ln
        def display(self) :
                print(' Name = ', self.firstname, self.lastname)

class Student (Person):
        """ Derived Class derived from Base Class Person"""
        pass    #No attributes and methods of derived class are defined

p1 = Person ('Py', 'Program')   #p1 is object of base class
p1.display( )

s1 = Student('Python', 'Programming')   # s1 is object of derived class
s1. display( )
```

Student class inherits all properties (attributes and methods) of base class Person.

➢ The scope of derived class is : Scope of it's own derived class followed by scope of the base class.
➢ Two types:
  - Single Inheritance
  - Multiple Inheritance

- Multilevel inheritance

Single Inheritance : Creating single derived class out of single base class. Scope of derived class is **Scope of its own derived class followed by scope of base class from which it is derived.**

Multiple Inheritance : Creating single derived class from multiple base classes. Scope of the derived class is **Scope of its own derived class followed by scope of base casses from which is derived in the order os representation from left to right.**

Example:

```
class Base1:
        def _ _init_ _(self):
                self.str1 = ‘Python’
                print(‘Base1’)

class Base2:
        def _ _init_ _(self):
                self.str2 = ‘Program’
                print(‘Base2’)

class Derived(Base1. Base2):
        def _ _init_ _(self):
                Base1._ _init_ _(self);
                Base2._ _init_ _(self);
                print(‘Derived’)
        def display(self):
                print(self.str1, self.str2)
```

```
D = Derived ( )
D.display( )
```

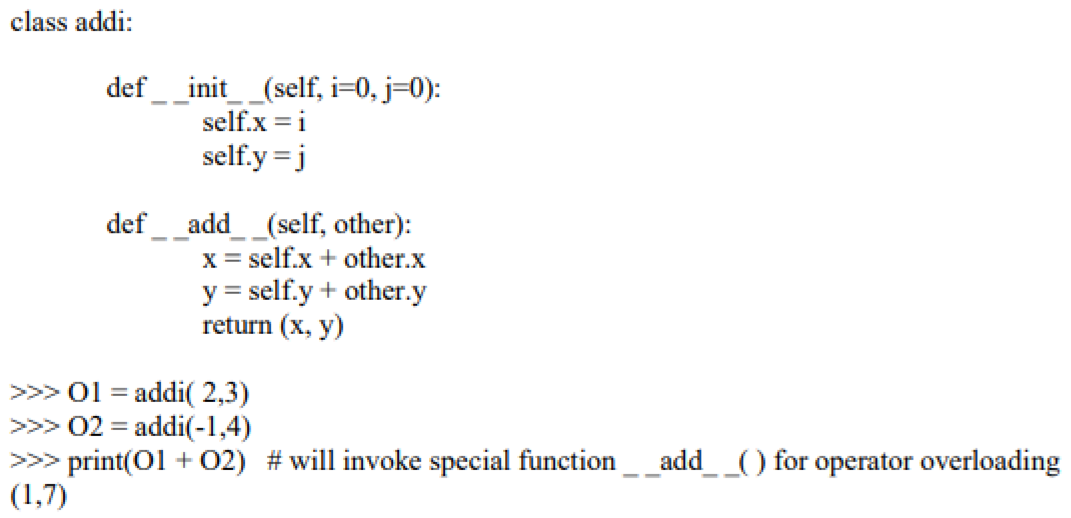
```
Ouptut:
Base1
Base2
Derived
Python Program
```

Multilevel inheritance : One base class creates one derived class and that derived class creates another derived class.The scope of the second derived class is its **own class followed by the scope of the Base Class**

```
class Base:
        def _ _init_ _(self, n):
                self.name = n
        def getn(self):
                return self.name

class Derived1(Base):
        def _ _init_ _(self, n, a):
                self.age = a
                Base._ _init_ _(self, n)
        def geta(self):
                return self.age

class Derived2(Derived1):
        def _ _init_ _(self, n, a, g):
                self.gender = g
                Derived1._ _init_ _(self, n, a)
        def getgen(self):
                return self.gender

D = Derived2(‘Py’, 10, ‘M’)
print(D.getn(), D.geta( ), D.getgen( ))
```

Polymorphism

- ➢ Single entity with multiple functionality.
- ➢ For example :
  - ○ Built-in keywords
    - ■ **len()** : which is used to get the **length of string** also returns the **number of elements in a list**.
    - ■ **+** : used to **concatenate two strings** also used as **addition operator for two integers.**
- ➢ Operator Overloading :
  - ○ Single operator multiple functionality.



```
class addi:

        def _ _init_ _(self, i=0, j=0):
                self.x = i
                self.y = j

        def _ _add_ _(self, other):
                x = self.x + other.x
                y = self.y + other.y
                return (x, y)

>>> O1 = addi( 2,3)
>>> O2 = addi(-1,4)
>>> print(O1 + O2)   # will invoke special function _ _add_ _( ) for operator overloading
(1,7)
```

PYTHON FILES

File Handling :

1. **File :**

File is a named location on the disc to store the information permanently in a non-volatile memory [ROM].

RAM is volatile memory which loses data when power is turned off.

In python, file operation takes place in the following order :

i. Open file (for read or write mode)

ii. Read or write operation

iii. Close the file.

Open File :

To open a file a built-in function **open()** is used. This function returns a file object or handle which will be used to read or modify the file.

syntax : file_handle = open('<file_path\\file_name>', '<mode>')

While giving path use **double backslash(\\)** as it is used as escaping character.

mode parameter is not mandatory, and types :

1. **'r'** - File is opened for read mode which is a default mode. It opens only if the file is present or else it raises an exception called "FileNotFound" error.
2. **'w'** - File is open for write mode. If a file is present, then it opens and the old data is overwritten by the new data. If not present then a new file is created and the data is written into it.
3. **'a'** - File is open for append mode. If a file is present then the data is added at the end of the file. If not present, then a new file is created and data is written into it.
4. **'r+'** - Opens in read and write mode.
5. **'x'** - Creates an exclusive file if not present. If present it returns an error.
6. **'b'** - Opens in binary mode. Non text file mode.

Read or Write Operation :

To write, mode : w, a or x

To read, mode : r or r+

Writing a string or sequence of bytes is done using the **write()** function which returns the number of characters or bits written into the file.

syntax : <file_handle>.write(<data_to_write>)

Reading operation is done in several methods

1. **read(<size>)** : Read size number of data. If size is not present, then read operation is done till the end of the file.
   Size is optional. If not given, then it reads entire file from start to end.

```
f.read(4) #reads first 4 characters in the sequence
f.read(4) #reads next 4 characters in the sequence
f.read( ) # reads remaining characters till the end of the file.
```

2. **readline()** : Reads individual lines of a file
3. **readlines()** : Reads all the lines in the file till end of file.

```
print(f.readline( ))  # reads and prints first line
print(f.readlines( )) #reads and prints the remaining lines
```

Close File :

When the file operations are done, the file needs to be closed which will free up some resources tied with the file. Inbuilt function **close()** is used to close the file.

syntax : file_handle.close()

- Using the **with** statement, it'll automatically close the file after doing the complete operation.

  syntax : with open(<filepath\\filename>,<mode>) as <file_handle> :

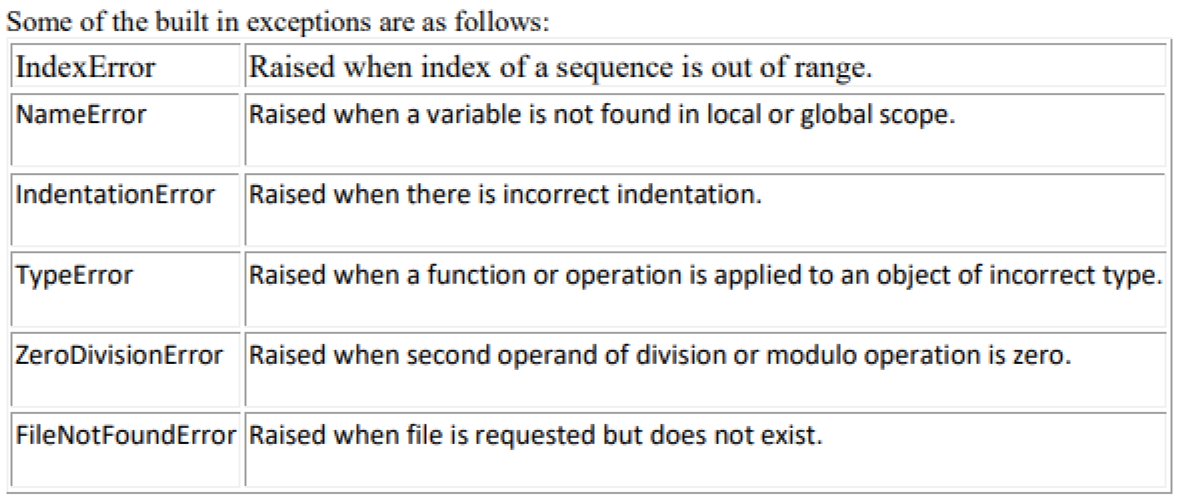
  #operations

- To check whether the file is still open or not use **closed** flag with the file_handle

  syntax : <file_handle>.closed

PYTHON EXCEPTIONS

- Runtime errors caused due to unusual conditions in the code.
- Python Interpreter raises the exception when it runs through unusual conditions in code.
- For eg.: DivideByZeroError, FileNotFoundError, IndexErroretc.



Some of the built in exceptions are as follows:

IndexError	Raised when index of a sequence is out of range.
NameError	Raised when a variable is not found in local or global scope.
IndentationError	Raised when there is incorrect indentation.
TypeError	Raised when a function or operation is applied to an object of incorrect type.
ZeroDivisionError	Raised when second operand of division or modulo operation is zero.
FileNotFoundError	Raised when file is requested but does not exist.

- Ways that python interpreter handles the exception.:
  - When an exception is raised by python, the execution the code gets terminated.
  - When an exception is raised by python, Exception Handling Mechanism must be provided.

Exception Handling:

Python provides handling of such exceptions (runtime errors) through Exception Handling Mechanism.

Many formats of **try & except** blocks are present below :

1. For handling individual exceptions.

   syntax :

```
try :
        #Set of operations to identify the exception.
        #If exception, then it is raised.
except <Excep_1> :
        #If the exception raised in try block matches <Excep_1>
        #Then it is caught and handled.
except <Excep_2> :
        #If the exception raised in try block matches <Excep_2>
        #Then it is caught and handled.
else :
```

```
                #If no exceptions are raised.
                #Execute these commands.
```

Example 1:

```
try:
   f = open('samp.txt','r')
   f.write(' Test File\n')
except IOError:
   print('Error: Cannot perform write operation as file opened in read mode')
else:
   print('Write successful')
   f.close( )
```

2. For handling all exceptions.

syntax :

```
        try :
                #Set of operations to identify the exception.
                #If exception, then it is raised.
        except :
                #If any exception is raised, then it is caught and handled.
        else :
                #If no exceptions are raised.
                #Execute these commands.
```

Example:

```
inp = input('Enter Temperature:')
try:
   f = float(inp)
   Celsius = (f – 32.0) * 5.0 / 9.0
   print(Celsius)
except:
   print('Enter a valid number')
```

3. For handling multiple exceptions.

syntax :

```
        try :
                #Set of operations to identify the exception.
                #If exception, then it is raised.
        except (<Excep_1>,<Excep_2>,<Excep_3>.......<Excep_N>)
                #If any of the above mentioned exceptions matches.
                #It is caught and handled.
        else :
                #If no exceptions are raised.
                #Execute these commands.
```

Example:

```
try:
   f=open('file.txt')
   f.readlines( )
except (IOError, FileNotFoundError):
   print('Read Unsuccessful')
else:
   print('Read Successful')
```

4. **try, finally** block.

   syntax :

```
        try :
            #Set of operations to identify the exception.
            #If exception, then it is raised.
        finally :
            #Statements are executed if or if not exceptions are raised.
```

```
while true:
        try:
                f=open('test.txt','w')
                f.write('Test File')
        finally:
                print('File Operation Done')
                f.close( )
```

User-defined Exceptions :

It can be created by using the concept of **class & inheritance**.

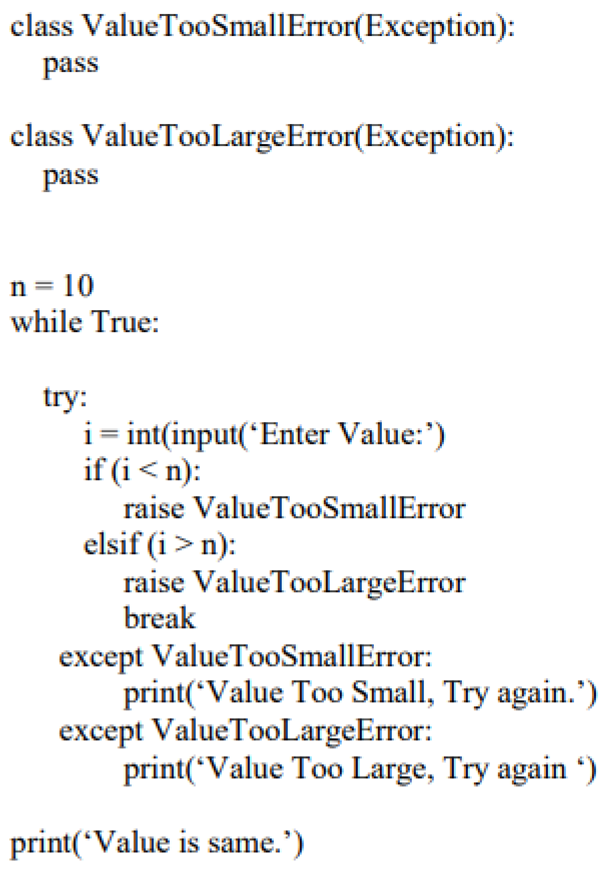
Exceptions can be generated by creating a new class which must be derived from an already existing base class called Exception class.

Example:

```
class ExceptionError(Exception):
    pass

raise ExceptionError   #raising an user defined exception manually
```

ExceptionError is the class which is derived from the **Exception** base class.

raise is a keyword which is used to raise the both builtin and user defined exceptions manually.



```
class ValueTooSmallError(Exception):
   pass

class ValueTooLargeError(Exception):
   pass


n = 10
while True:

   try:
      i = int(input(‘Enter Value:’)
      if (i < n):
         raise ValueTooSmallError
      elsif (i > n):
         raise ValueTooLargeError
         break
    except ValueTooSmallError:
         print(‘Value Too Small, Try again.’)
    except ValueTooLargeError:
         print(‘Value Too Large, Try again ‘)

print(‘Value is same.’)
```

Python games :

- https://www.codemonkey.com/courses/banana-tales/
- https://www.codewars.com/collections/basic-python
- https://www.codecombat.com/play
- https://www.py.checkio.org