Introduction to Python (Revision Guide)

Submited By:
Farhan Anjum
Abdul Rehman
Hafiz Waleed
Muhammad Habib
Ahmed Arsam (prof reader)
Ali Raza (Team Leader)
Zeeshan Hameed

Senior Instructor:
Mr. Salman Mahmood Qazi
Junior Instructor:
Mr. Muhammad Owais

Al-Khwarizmi Institute of Computer Science (KICS)
University of Engineering and Technology, G.T. Road,
Lahore, Pakistan
2022

بسم الله الرحمن الرحيم

DEDICATION

I dedicate all my efforts and struggles of the educational life to my dear parents and family members; without them I am meaningless.

Also I devote the work of this Revision Guide to respectable and honorable teachers who taught and supported me in developing my personality as a competent professional.

ACKNOWLEDGEMENT

I am grateful to Almighty ALLAH for giving me the strength, knowledge and understanding to complete this project. His love has been more than sufficient to keep and sustain me.

I would like to special thanks senior Instructor **Mr. Salman Mahmood Qazi** (Lecturer at Physics Department, GCU Lahore.) who helped me to the completion of this Revision Guide.

My thanks also extend to my Junior Instructor, **Mr. Muhammad Owais**, (Research Assisstant at UET Lahore) who patiently guided me through the entire course and completion of my Revision Guide.

I also extend gratitude and appreciation to my all classmates in the department who have taught me at one point or the other. May ALLAH continue to bless, protect and guide them.

My profound gratitude goes to my wonderful lovely elder brother, **Mr. Muhammad Luqman** for his valuable support, patience, time and guidance in seeing me to the completion of this Data Science course.

I also wish to acknowledge the great support of my parents, siblings and friends who have been a source of inspiration towards my academic pursuit. ALLAH blesses you all.

Zeeshan Hameed

Table of Content

Introduction to Python

1stWeek lectures:
Submitted to: Mr. Salman Mahmood Qazi & Muhammad Owais
Farhan Anjum
Fanjum524@gmail.com

What is Python?

Python is a high-level popular programming language. It was created by Guido van Rossum, and released in 1991.
It is used for:
web development (server-side),
software development,
mathematics (Numpy, Sci-py).
Data Science and Machine Learning.

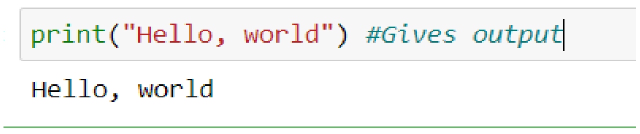
Getting Started with Python.

What is IDE?

IDE stands for Integrated Development Environment, a software used to develop programs.
Example:
Google Colaboratory
Anaconda (Jupyter Notebook)
Pycharm.
Here, preferably, we are using the Anaconda (Jupyter Notebook).
How to install the Anaconda?
Download Anaconda. We recommend downloading Anaconda's latest Python 3 version (recommended Python 3.5)

HOW TO INSTALL:

Install the version of Anaconda that you downloaded, following the instructions on the download page.
Congratulations, you have installed Jupyter Notebook. To run the notebook.
Syntax:
The syntax is the grammatical rules of a programing language to write the code.
Example:
Print (“Hello World”)
OUTPUT
Hello, Word

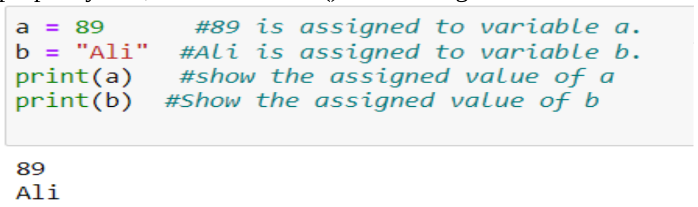


```
print("Hello, world") #Gives output
```

```
Hello, world
```

The above-mentioned example will print the Hello World, with given syntax,

if we write:
Print “Hello world”
#wrong Syntax
It would give a syntax error as we have not mentioned the proper syntax, the brackets or () are missing.

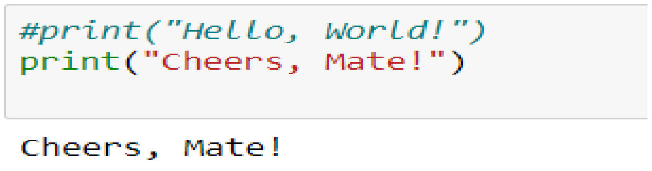


```
a = 89      #89 is assigned to variable a.
b = "Ali"   #Ali is assigned to variable b.
print(a)    #show the assigned value of a
print(b)   #Show the assigned value of b
```

```
89
Ali
```

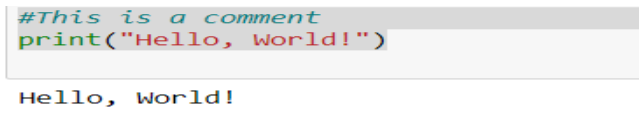
Python Comments:

Comments are the non-executable chunk of the words in code, to explain the Python code.
Comments can be used to make the code more readable.
Comments can be used to prevent execution when testing code.
Comments start with a #, and Python will ignore them.



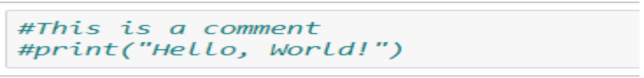
```
#print("Hello, World!")
print("Cheers, Mate!")
```

```
Cheers, Mate!
```



```
#This is a comment
print("Hello, World!")
```

```
Hello, World!
```



```
#This is a comment
#print("Hello, World!")
```

Example:
# This is the comment
print ("Hello, World!")
“This is a comment” is non-executable as it’s written with #, while the lower part of the code will give output: Hello, World.
A comment does not have to be text that explains the code, it can also be used to prevent Python from executing code.
Example:
#print("Hello,World!")
print("Cheers, Mate!")
In this example, the code with #, will not execute while the lower part will print Cheers, Mate!

Python Variables:

A variable is the name of a reserved area allocated in memory. In other words, it is the name of the memory location for storing data.
Example:
A = 89
#89 is assigned to variable a.
b = "Ali"
#Ali is assigned to variable b.
print(a)
#show the assigned value of a
print(b)
#Show the assigned value of b
‘A’ is a variable whose value is 89, and ‘b’ is also a variable having the value ALI
Variable names are **case-sensitive.**
Example:
A = 55 #Uppercase A is different variable
a = 55 #Lower case a is a different variable
Both are two different variables.

Python Data Types:

In python, data types are very important, Variables can store data of different types.

These are the built-in Data types in Python.

Literals are static values, which can be integer, float, or complex numbers.

- String Literals
- Numeric Types
- Boolean Type

String Type: Anything was written in inverted commas, the python will take them as string data type.

Example:

A = “Pakistan”

B= “55”

In both examples, the python will take them as strings, although 55 is numeric.

Numeric Types: The numeric data type includes integer, float, and complex.

A = int (5)

B = float (2.5)

C = 4i + 5 is a complex number.

Boolean Type:

Booleans represent one of two values: True or False.

Example:

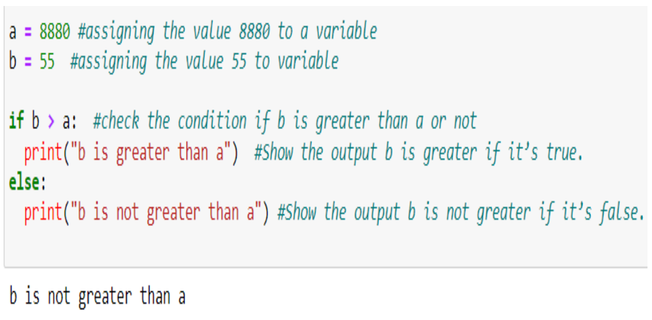
a = 8880

#assigning the value 8880 to a variable

b = 55

#assigning the value 55 to variable

if b > a:



```
a = 8880 #assigning the value 8880 to a variable
b = 55  #assigning the value 55 to variable

if b > a:  #check the condition if b is greater than a or not
  print("b is greater than a")  #Show the output b is greater if it’s true.
else:
  print("b is not greater than a") #Show the output b is not greater if it’s false.
```

```
b is not greater than a
```

#check the condition if b is greater than a or not

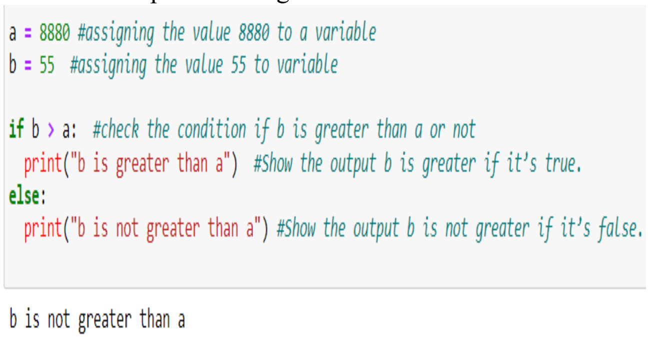
print("b is greater than a")

#Show the output b is greater if it’s true.

else:

print("b is not greater than a")

#Show the output b is not greater if it’s false.



```
a = 8880 #assigning the value 8880 to a variable
b = 55  #assigning the value 55 to variable

if b > a:  #check the condition if b is greater than a or not
  print("b is greater than a")  #Show the output b is greater if it’s true.
else:
  print("b is not greater than a") #Show the output b is not greater if it’s false.
```

```
b is not greater than a
```

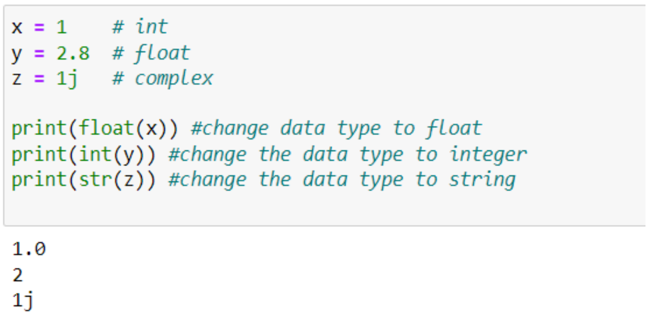
In the above example, we are comparing two variables and checking whether the condition is true or false. As we know the value in variable is greater than the value in variable b, so the condition is true.

Type Casting:

The process of converting the value of one data type (integer, string, float, etc.) to another data type is called typecasting.

Example:

Data Type can be inferred using the function Type ().



```
x = 1    # int
y = 2.8  # float
z = 1j   # complex

print(float(x)) #change data type to float
print(int(y)) #change the data type to integer
print(str(z)) #change the data type to string
```

```
1.0
2
1j
```

x= 1   #int

y= 2.8 #float

z = 1j   # complex

print(type(x))

#show the type of value assigned to variable x

print(type(y))

#show the type of value assigned to variable y

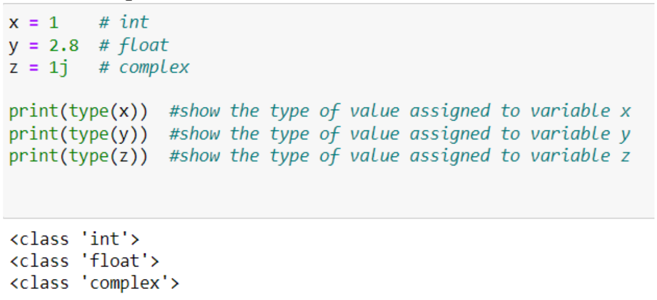
print(type(z))

#show the type of value assigned to variable z

When we execute the above code, it will change the data type of each literal, say it’ll change Int to float, float to Int, and complex to str.

Type Inference:

When we execute the above code, it will give the data type of each variable, for x it’ll be Int, for y it’ll be float, and z it’ll be complex.



```
x = 1    # int
y = 2.8  # float
z = 1j   # complex

print(type(x))  #show the type of value assigned to variable x
print(type(y))  #show the type of value assigned to variable y
print(type(z))  #show the type of value assigned to variable z
```

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

Python Operators:

Operators are used to performing operations on variables and values.

There are four basic operators In Python.

1. Arithmetic operators
2. Assignment operators
3. Comparison operators
4. Logical operators

Arithmetic Operators:

Arithmetic operators are used with numeric values to perform common mathematical operations:

There are six basic arithmetic operations in Python:

Addition (+)

Addition operator is used to add the 2 or more value

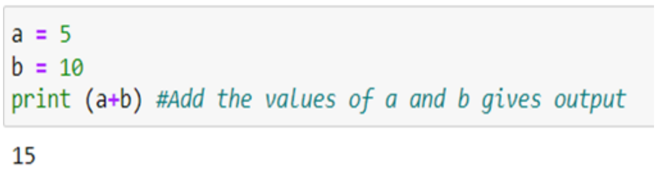
Example

a = 5

b = 10

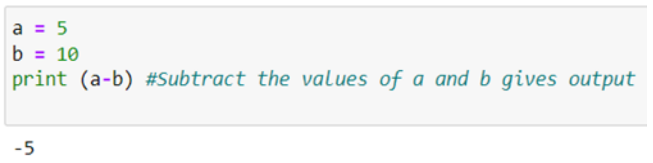
print (a+b)

#Add the values of a and b gives output.
In this example, the output will be 15, as it has added both numbers.



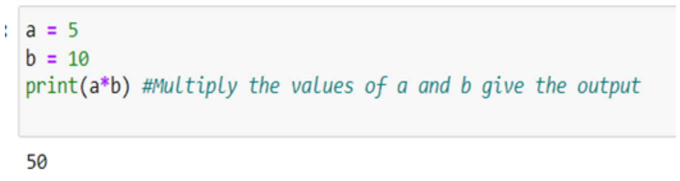
```
a = 5
b = 10
print (a+b) #Add the values of a and b gives output

15
```

Subtraction (-):
a = 5
b = 10
print (a-b)
#Subtract the values of a and b gives output
In this example, the output will be 5, as it has subtracted both numbers.



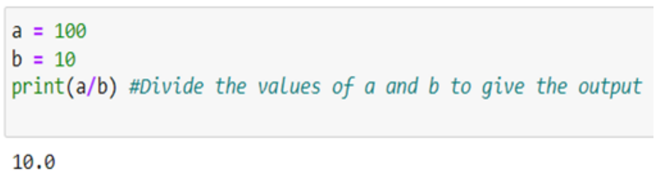
```
a = 5
b = 10
print (a-b) #Subtract the values of a and b gives output

-5
```

Multiplication (*):
a = 5
b = 10
print(a*b)
#Multiply the values of a and b to give the output
In this example, the output will be 50, as it has multiplication of both numbers.



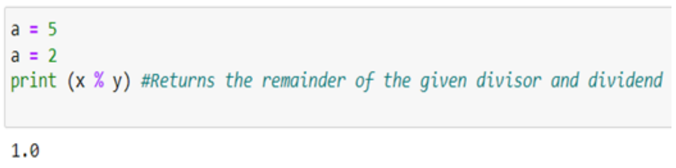
```
a = 5
b = 10
print(a*b) #Multiply the values of a and b give the output

50
```

Division (/):
a = 100
b = 10
print(a/b)
#Divide the values of a and b to give the output
In this example, the output will be 10, as it has multiplication of both numbers.



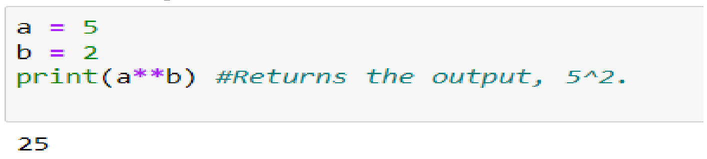
```
a = 100
b = 10
print(a/b) #Divide the values of a and b to give the output

10.0
```

Modulus (%)
a = 5
a = 2
print (x % y) #Returns the remainder of the given divisor and dividend
In this example, the output will be 1, as it gives a remainder of divided numbers.



```
a = 5
a = 2
print (x % y) #Returns the remainder of the given divisor and dividend

1.0
```

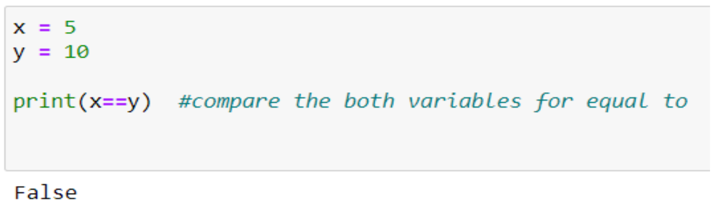
Exponentiation (**):
a = 5
b = 2
print(a**b)
#Returns the output, 5^2.



```
a = 5
b = 2
print(a**b) #Returns the output, 5^2.

25
```

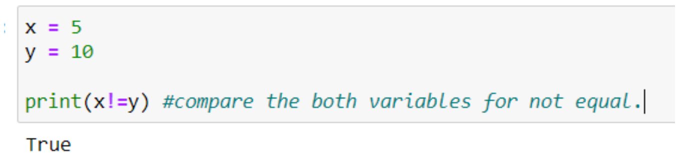
In this example, the output will be 25, as it is multiplied by 5 powers 2.

Assignment operators:
The assignment operator assigns the value to the given variable. It is represented with this sign "="
Example:
x = 5
y = 12
a = "ALI"
x = x + 5, x+=5
x = x - 5, x-=5
x = x * 5, x*=5
x = x / 5, x/=5
Example
X = 0 #Initializing the value of x
while >=20: #Defining the condition
    print x
    x=x+2,
#gives the output with increment of 2.



```
x = 5
y = 10

print(x==y)  #compare the both variables for equal to

False
```

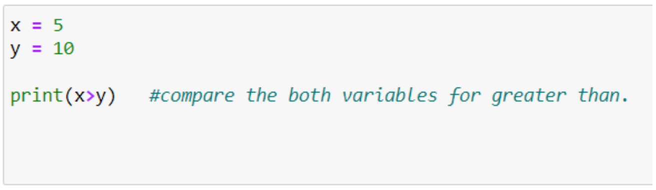
Comparison operators: This operator is used to compare two values.
There are six assignment operators:
Equal to (==):
x = 5
y = 10
Print(x==y)



```
x = 5
y = 10

print(x!=y) #compare the both variables for not equal.

True
```

#compare the both variables for equal to
The output will be FALSE.
Not equal (!=)
x = 5
y = 10
Print(x!=y)
#compare the both variables for not equal
The output will be TRUE.
Greater than (>)
x = 5
y = 10
Print(x>y)
#compare both variables for greater than.

The output will be FALSE.



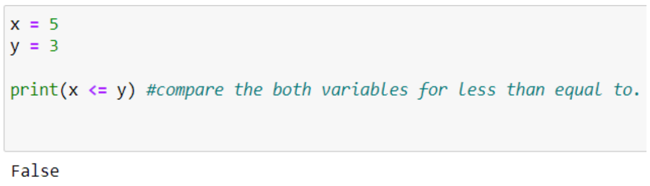
```
x = 5
y = 10

print(x>y)   #compare the both variables for greater than.
```

Less than (<):

x = 5
y = 10
Print(x<y)
#compare both variables for less than.
The output will be TRUE

Less than or equal to (<=):

x = 5
y = 3
print (x <= y)



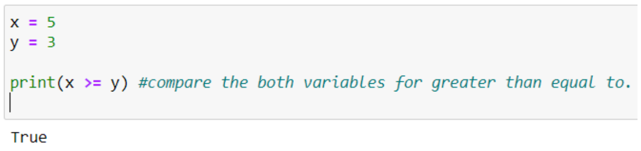
```
x = 5
y = 3

print(x <= y) #compare the both variables for less than equal to.
```

```
False
```

#compare the both variables for less than equal to Returns False because 5 is neither less than nor equal to 3

Greater than or equal to (>=):

x = 5
y = 3
print(x >= y)
#compare the both variables for greater than equal to



```
x = 5
y = 3

print(x >= y) #compare the both variables for greater than equal to.
```

```
True
```

Returns True because five is greater, neither equal to.

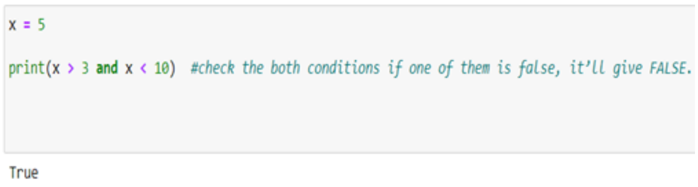
Logical operators:

Logical operators are used to combining two conditional statements.
There are three basic Logical operators.

- AND
- OR
- NOT

AND Operator:

Example:
x = 5
print (x > 3 and x < 10) #check the both conditions if one of them is false, it'll give FALSE.
Returns True because 5 is greater than 3 AND 5 is less than 10.

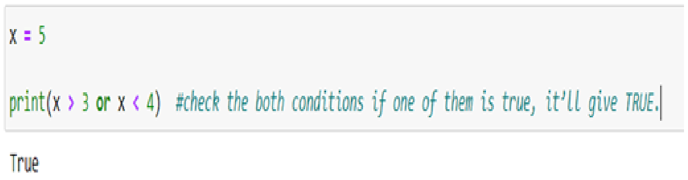


```
x = 5

print(x > 3 and x < 10)  #check the both conditions if one of them is false, it'll give FALSE.
```

```
True
```

OR Operator:

x = 5
print (x > 3 or x < 4) #check the both conditions if one of them is true, it'll give TRUE.
Returns True because one of the conditions is true (5 is greater than 3, but 5 is not less than 4)

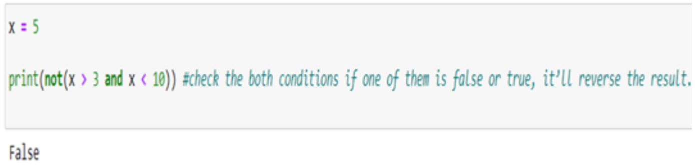


```
x = 5

print(x > 3 or x < 4)  #check the both conditions if one of them is true, it'll give TRUE.
```

```
True
```

NOT Operator:

x = 5
print(not(x > 3 and x < 10)) #check the both conditions if one of them is false or true, it'll reverse the result.
Returns False because not is used to reverse the result



```
x = 5

print(not(x > 3 and x < 10)) #check the both conditions if one of them is false or true, it'll reverse the result.
```

```
False
```

Introduction to Built-In Data Types in Python

List, Tuple, Set, and Dictionary
Name: Abdur Rahman
Date of Lecture: March 2022
Date of performance: March 2022
Date of submission: May 13, 2022
Submitted to: Sir Salman Mahmood Qazi & Sir Owais Saleem
Abdulrehman100198@gmail.com

Introduction

In programming, data types is an important concept. Variable can store different type of data and different types can do different things.

Here we will study Set, List, Tuple, and Dictionary.

Python Sets:

In Python, **Set** is an unordered collection of data type that is iterable, mutable and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements.

The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.

Characteristics

i- Sets are unordered

ii- Set is unindexed

iii- There is no way to change the items in sets

iv- Sets cannot contain duplicate values

v- Sets use curly brackets. ‘{}’

Set Methods

Let’s, consider the following set:

S = {1,8,2,3}

i- Len(S)

Returns 4, the length of the set.

ii- S.remove(8)

Update the set S and remove the 8.

iii- S.pop()

Return the arbitrary element from the set and return the element removed.

iv- S.clear()

Empty the Set S.

Examples

i- Simple Code

```
    set = {"apple", "banana", "cherry"}
    print(set)
    # Note: the set list is unordered, meaning: the
items will appear in a random order.
Output:
{'banana', 'cherry', 'apple'}
```

ii- Duplicate values in set will ignored

```
    set = {"apple", "banana", "cherry", "apple"}
    # here apple is used twice therefore it will be
ignored in output
print(set)
output:
{'banana', 'cherry', 'apple'}
```

iii- Get the Length of a Set

```
    Thisset = {"apple", "banana", "cherry"}
print(len(thisset))
output:
3
```

iv- Set Items - Data Types

Set items can be of any data type:

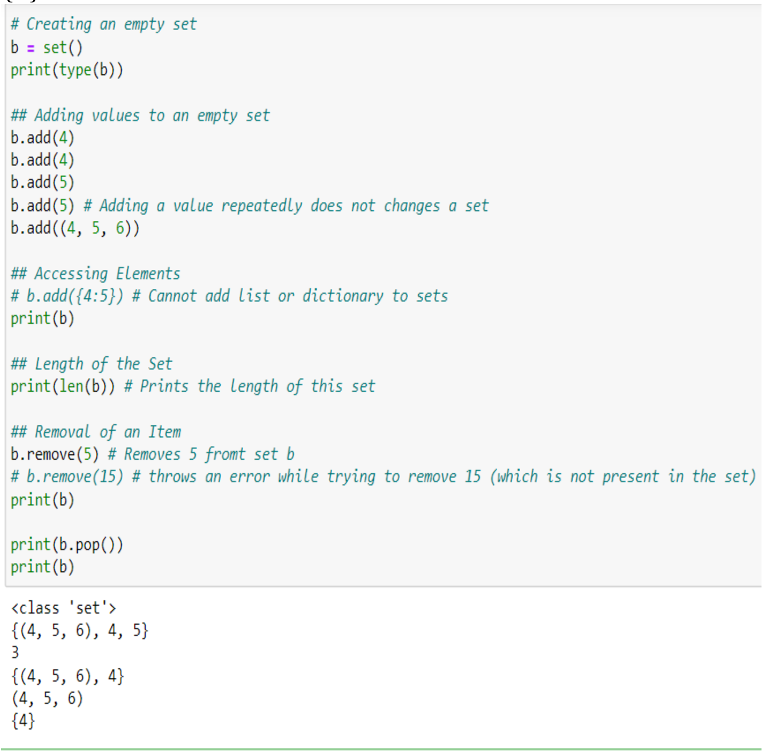
```
set1 = {"apple", "banana", "cherry"}
# String
set2 = {1, 5, 7, 9, 3}
# integer
set3 = {True, False, False}
# Boolean
print(set1)
print(set2)
print(set3)
output:
    {'cherry', 'apple', 'banana'}
    {1, 3, 5, 7, 9}
    {False, True}
```

A set can contain different data types:

```
    set1 = {"abc", 34, True, 40, "male"}
print(set1)
Output
{True, 34, 40, 'male', 'abc'}
```

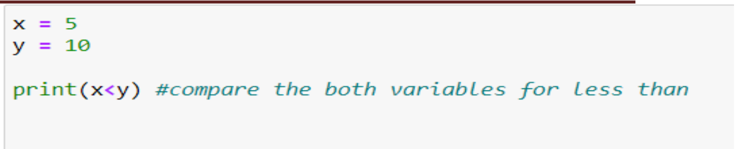
vi- Complete Python code:

```
# Creating an empty set
b = set()
print(type(b))
## Adding values to an empty set
b.add(4)
b.add(4)
b.add(5)
b.add(5) # duplicate value cannot add in a set
b.add((4, 5, 6))
## Accessing Elements(indexing)
# b.add({4:5}) # Cannot add list or dictionary to sets
print(b)
output
<class 'set'>
{(4, 5, 6), 4, 5}
## Length of the Set
print(len(b)) # Prints the length of this set
output
<class 'set'>
3
## Removal of an Item
b.remove(5) # Removes 5 fromt set b
print(b)
output
<class 'set'>
{(4, 5, 6), 4}
# b.remove(15) # throws an error while trying to remove
15 (which is not present in the set)
print(b.pop())
print(b)
output
 (4, 5, 6)
{4}
```



```
# Creating an empty set
b = set()
print(type(b))

## Adding values to an empty set
b.add(4)
b.add(4)
b.add(5)
b.add(5) # Adding a value repeatedly does not changes a set
b.add((4, 5, 6))

## Accessing Elements
# b.add({4:5}) # Cannot add list or dictionary to sets
print(b)

## Length of the Set
print(len(b)) # Prints the length of this set

## Removal of an Item
b.remove(5) # Removes 5 fromt set b
# b.remove(15) # throws an error while trying to remove 15 (which is not present in the set)
print(b)

print(b.pop())
print(b)
```

```
<class 'set'>
{(4, 5, 6), 4, 5}
3
{(4, 5, 6), 4}
(4, 5, 6)
{4}
```



```
x = 5
y = 10

print(x<y) #compare the both variables for less than
```

# Creating a Set
# A set is created by using curly brackets {} .
# The objects are placed inside those brackets and are separated by commas (,).
pets={"dog", "cat", "rabbit"}
print (pets)
print(type(pets))
Output
{'rabbit', 'cat', 'dog'}
<class 'set'>
vii-

# Accessing Items
# Unlike lists and tuples, you can NOT access the item s of a set using indexes.
# It is because a set is unordered and not indexed.
# However, we can use the for loop to access all its ite ms one-by-one.

```
pets = {"dog", "cat", "rabbit"}
for pet in pets:
   print(pet)
```

Output

```
rabbit
cat
dog
```

Python List

Lists are used to store multiple items in a single variable.
Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are **Tuple**, **Set**, and **Dictionary**, all with different qualities and usage.
Lists are created using square brackets []
Example:
thislist = ["apple", "banana", "cherry"]
print(thislist)
Output

```
['apple', 'banana', 'cherry']
```

Characteristics

i- List items that are ordered, changeable, and allow duplicate values.

ii- List items are indexed, the first item has index [0], the second item has index [1], etc.

iii- The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Note: There are some **list methods** that will change the order, but in general: the order of the items will not change.

List Methods

Consider the following list

L1= [1,5,7,4,8,2]

i- L1.sort

Update list to [1,2,4,5,7,8]

ii- L1.reverse

Update to [2, 8, 4, 7, 5, 1]

iii- L1.append(9)

Add 9 in the end of the list and update to [1, 5, 7, 4, 8, 2, 9]

iv- L1.insert(3,0)

This will add 0 at 3 index and update list to [1, 5, 7, 0, 4, 8, 2]

v- L1.pop(3)

This will delete the element at 3 index and return its value and update list to [1, 5, 7, 8, 2]

vi- L1.remove(7)

This will remove the 7 from the list and update the list to [1, 5, 4, 8, 2]

Examples

i- Allow Duplicate

thislist = ["apple", "banana", "cherry", "apple", "cherry"]
print(thislist)
Output

```
['apple', 'banana', 'cherry', 'apple', 'cherry']
```

ii- All List Methods

```
# Create a list using []
a = [1, 2 , 4, 56, 6]
# Print the list using print() function
print(a)
  Output
  [1, 2, 4, 56, 6]
  #reverse the value of list using a.reverse
a.reverse()
print(a)
  Output
  [6, 56, 4, 2, 1]
#arrange the element in the list using sort
a.sort()
print(a)
  Output
  [1, 2, 4, 6, 56]
# Access using index using a[0], a[1], a[2]
print(a[2])
Output
4
#update the value of list using append
a.append(32)
print(a)
Output
[1, 2, 4, 6, 56, 32]
# Change the value of list using insert
a.insert(3,33)
print(a)
Output
[1, 2, 4, 33, 6, 56, 32]
# We can create a list with items of different types
c = [45, "DATA", False, 6.9]
print(c)
Output
  [45, 'DATA', False, 6.9]
```

iii- Storing values in list by taking data from the user

```
# storing the values of fruits by getting values from user
f1 = input("Enter Fruit Number 1: ")
f2 = input("Enter Fruit Number 2: ")
f3 = input("Enter Fruit Number 3: ")
f4 = input("Enter Fruit Number 4: ")
f5 = input("Enter Fruit Number 5: ")
f6 = input("Enter Fruit Number 6: ")
f7 = input("Enter Fruit Number 7: ")
myFruitList = [f1, f2, f3, f4, f5, f6, f7]
print(myFruitList)
```

Output

```
Enter Fruit Number 1: apple
Enter Fruit Number 2: banana
Enter Fruit Number 3: orange
Enter Fruit Number 4: kiwi
Enter Fruit Number 5: pine apple
Enter Fruit Number 6: grapes
Enter Fruit Number 7: peach
['apple', 'banana', 'orange', 'kiwi', 'pine apple', 'grapes', 'peach']
```

Python Tuples

Tuples are used to store multiple items in a single variable.

Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.

A tuple is a collection that is ordered and unchangeable.

Tuples are written with round brackets ().

Characteristics

i- Tuple items are ordered, unchangeable, and allow duplicate values.

ii- Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

iii- When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

iv- Tuples are unchangeable, meaning that we cannot change, add, or remove items after the tuple has been created.

v- It also allows duplicates. Since tuples are indexed, they can have items with the same value.

Tuple Methods

Consider the following tuple

```
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
```

i- Count() Returns the number of times a specified value occurs in a tuple

```
x = thistuple.count(5)
print(x)
```

Output

```
2
```

ii- Index() Searches the tuple for a specified value and returns the position of where it was found

```
x = thistuple.index(8)
print(x)
```

Output

```
3 # that means, position of 8 is at index 3.
```

Examples

i- Access Tuple Items

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

Output

```
Banana
```

Note: The first item has index 0.

ii- Negative Indexing

a. Negative indexing means start from the end.

b. -1 refers to the last item, -2 refers to the second last item etc.

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

Output

```
Cherry
```

iii- Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new tuple with the specified items.

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
#This will return the items from position 2 to 5.
#Remember that the first item is position 0,
#and note that the item in position 5 is NOT included
```

Output

```
('cherry', 'orange', 'kiwi')
```

Note: The search will start at index 2 (included) and end at index 5 (not included).

By leaving out the start value, the range will start at the first item:

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[:4])
#in this code the the range start from 0 index.
```

Output

```
('apple', 'banana', 'cherry', 'orange')
```

iv- Len() function

```
# Getting the Length of a Tuple To get the length or the number of items in a
pets = ("dog", "cat", "rabbit")
print(len(pets))
# tuple, use the len() method.
```

Output

```
3
```

v-

```
# combine tuple
pets = ("dog", "cat", "rabbit")
pets1=("thaing","pakistan","lahore")
pet_3=pets+pets1
print(pet_3)
```

Output

```
('dog', 'cat', 'rabbit', 'thaing', 'pakistan', 'lahore')
```

Python Dictionary

Dictionaries are used to store data values in key:value pairs.
A dictionary is a collection which is ordered*, changeable and do not allow duplicates.
Written in {}
As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

Characteristics

i- It is mutable

ii- It is ordered

iii- It cannot contain duplicate values

Dictionary Methods

Consider the following dictionary

```
A ={ “name” : “xyz”
  “from” : “ABC”
“marks” : [99,90,98] }
```

i- clear()

Removes all the elements from the dictionary

```
car =    {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
car.clear()
print(car)
Output
{}
```

ii- copy()

Returns a copy of the dictionary

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.copy()
print(x)
Output
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

iii- fromkeys()

Returns a dictionary with the specified keys and value

```
x = ('key1', 'key2', 'key3')
y = 0
thisdict = dict.fromkeys(x, y)
print(thisdict)
Output
{'key1': 0, 'key2': 0, 'key3': 0}
```

iv- get()

Returns the value of the specified key

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.get("model")
print(x)
output
Mustang
```

v- items()

Returns a list containing a tuple for each key value pair

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.items()
print(x)
dict_items([('brand', 'Ford'),
Output
 ('model', 'Mustang'), ('year', 1964)])
```

vi- keys()

Returns a list containing the dictionary's keys

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.keys()
print(x)
dict_keys(['brand', 'model', 'year'])
```

vii-pop()

Removes the element with the specified key

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
car.pop("model")
print(car)
Output
{'brand': 'Ford', 'year': 1964}
```

viii- popitem()

Removes the last inserted key-value pair

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
car.popitem()
print(car)
output
{'brand': 'Ford', 'model': 'Mustang'}
```

ix- update()

Updates the dictionary with the specified key-value pairs

```
car = {
  "brand": "Ford",
  "model": "Mustang",
   "year": 1964
}
car.update({"color": "White"})
print(car)
```

Output
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'White'}

x- values()

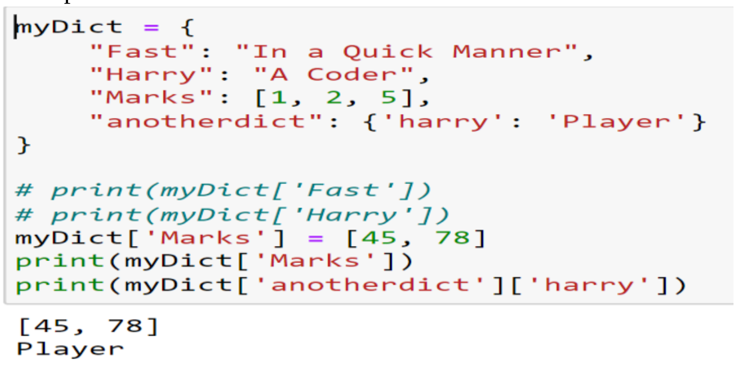
Returns a list of all the values in the dictionary

```
    car = {
      "brand": "Ford",
      "model": "Mustang",
      "year": 1964
    }
    x = car.values()
print(x)
```

Output

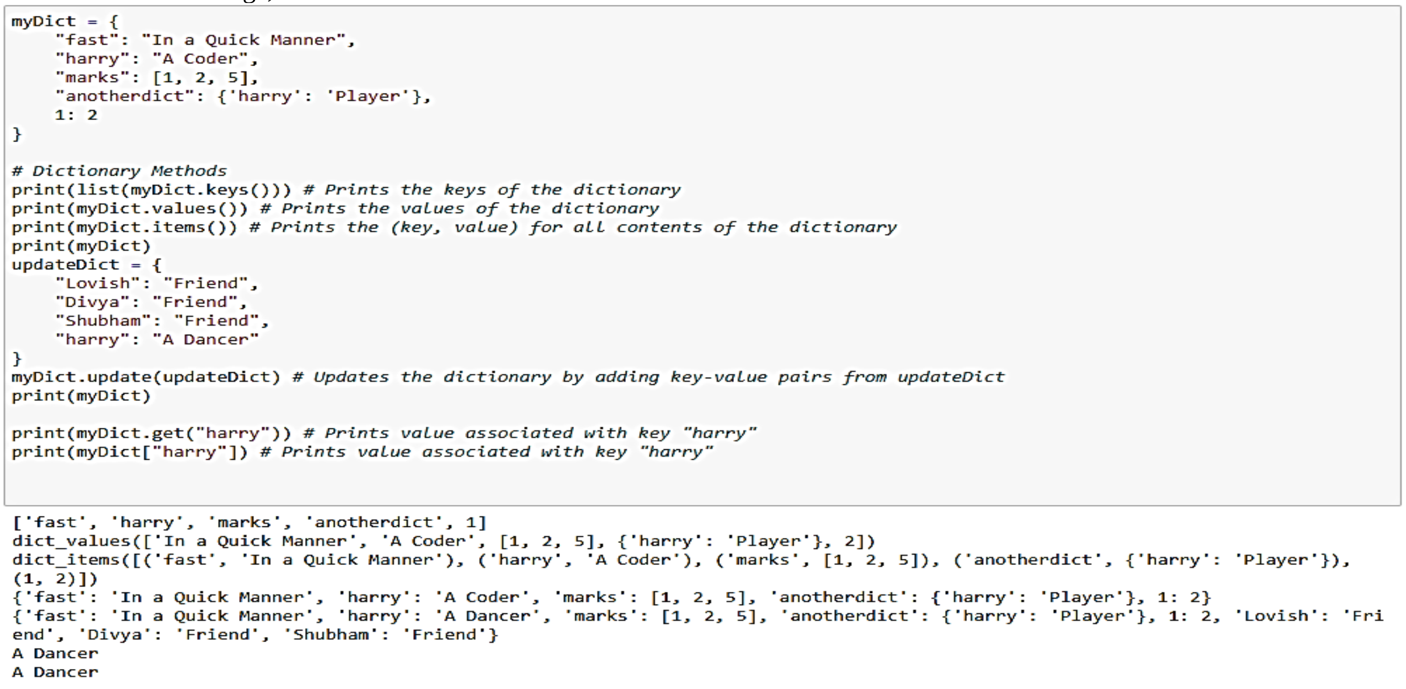
```
    car = {
      "brand": "Ford",
      "model": "Mustang",
      "year": 1964
    }
    x = car.values()
print(x)
```

Examples



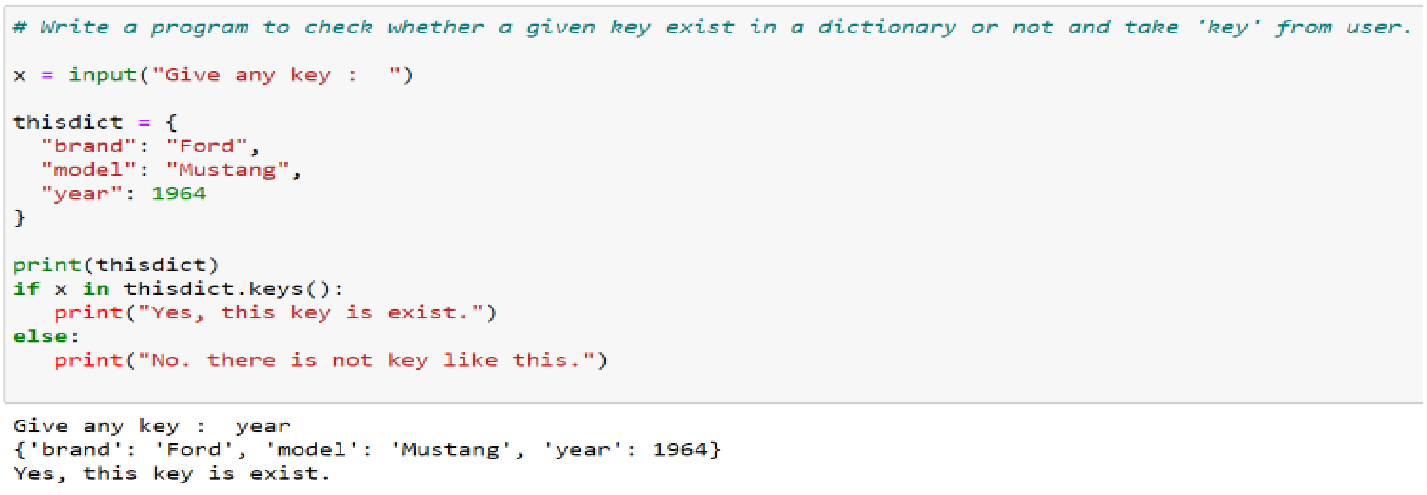
```
myDict = {
      "Fast": "In a Quick Manner",
      "Harry": "A Coder",
      "Marks": [1, 2, 5],
      "anotherdict": {'harry': 'Player'}
}

# print(myDict['Fast'])
# print(myDict['Harry'])
myDict['Marks'] = [45, 78]
print(myDict['Marks'])
print(myDict['anotherdict']['harry'])
```

```
[45, 78]
Player
```



```
myDict = {
    "fast": "In a Quick Manner",
    "harry": "A Coder",
    "marks": [1, 2, 5],
    "anotherdict": {'harry': 'Player'},
    1: 2
}

# Dictionary Methods
print(list(myDict.keys())) # Prints the keys of the dictionary
print(myDict.values()) # Prints the values of the dictionary
print(myDict.items()) # Prints the (key, value) for all contents of the dictionary
print(myDict)
updateDict = {
    "Lovish": "Friend",
    "Divya": "Friend",
    "Shubham": "Friend",
    "harry": "A Dancer"
}
myDict.update(updateDict) # Updates the dictionary by adding key-value pairs from updateDict
print(myDict)

print(myDict.get("harry")) # Prints value associated with key "harry"
print(myDict["harry"]) # Prints value associated with key "harry"
```

```
['fast', 'harry', 'marks', 'anotherdict', 1]
dict_values(['In a Quick Manner', 'A Coder', [1, 2, 5], {'harry': 'Player'}, 2])
dict_items([('fast', 'In a Quick Manner'), ('harry', 'A Coder'), ('marks', [1, 2, 5]), ('anotherdict', {'harry': 'Player'}),
(1, 2)])
{'fast': 'In a Quick Manner', 'harry': 'A Coder', 'marks': [1, 2, 5], 'anotherdict': {'harry': 'Player'}, 1: 2}
{'fast': 'In a Quick Manner', 'harry': 'A Dancer', 'marks': [1, 2, 5], 'anotherdict': {'harry': 'Player'}, 1: 2, 'Lovish': 'Fri
end', 'Divya': 'Friend', 'Shubham': 'Friend'}
A Dancer
A Dancer
```



```
# Write a program to check whether a given key exist in a dictionary or not and take 'key' from user.

x = input("Give any key :  ")

thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

print(thisdict)
if x in thisdict.keys():
    print("Yes, this key is exist.")
else:
    print("No. there is not key like this.")
```

```
Give any key :  year
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
Yes, this key is exist.
```

If else & elif Conditions

3^{rd} Week lectures:
Submitted by: Hafiz Waleed Ahmed
Submitted to: Mr. Salman Mahmood Qazi & Muhammad Owais
hafizwaleedahmeddatascience@gmail.com

What are Conditional Statements in Python?

Decision-making in a programming language is automated using conditional statements, in which Python evaluates the code to see if it meets the specified conditions. The conditions are evaluated and processed as true or false. If this is found to be true, the program is run as needed.

Example:

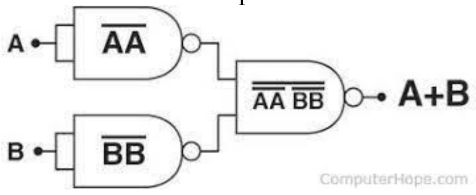
That is to say, to compare two values, which is greater and which is smaller. Depend on the condition.

Introduction to Boolean Values

The Python Boolean type has only two possible values True and False. Use Bool () function to test if a value is True or False.

Logical Operators

A logical operation is a special symbol or word that connects two or more phrases of information.



Types of logical operators

There are three logical operators:
And operator
Or operator
Not operator

And Operator

The and operator take two values, value 1 and value 2 follow the table value to find the result.

Boolean Value 1	Boolean Value 2	Result Boolean Value
True	True	True
True	False	False
False	True	False
False	False	False

Or Operator

The or Operator is similar to the and operator, but instead it implements the following table to find the result when given two Value 1 and Value 2

Boolean Value 1	Boolean Value 2	Result Boolean Value
True	True	True
True	False	True
False	True	True
False	False	False

Not Operator

The not Operator take a value, looks at its Boolean value and converts this value to its value to its Boolean counterpart. e.g. not True gives False, and not False gives True.

Python supports the usual logical conditions from mathematics:

Equals: a = = a

If the values of two operands are equal, then the condition becomes true. Otherwise, it will be a False.

Example: 2 = = 2

This is Condition True.

Not Equals: a ! = b

If the values of two operands are equal, then the condition becomes false. Otherwise, it will be true.

Example: 2! = 5

This Condition is True.

Less than: a < b

If the value of the left operand is less than the value of the right operand, then the condition becomes true.

Example: 2 < 5

This Condition is True.

Less than or equal to: a <= b

If the value of the left operand is less than or equal to the value of the right operand, then the condition becomes true.

Example: 2 <= 5

This condition is True.

Greater than: a > b

If the value of the left operand is greater than the value of the right operand, then the condition becomes true.

Example: 2 > 5

This condition is False.

Greater than or equal to: a >= b

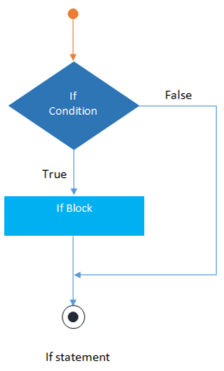
If the value of the left operand is greater than or equal to the value of the right operand, then the condition becomes true.

Example: 2 <= 5

This condition is True.

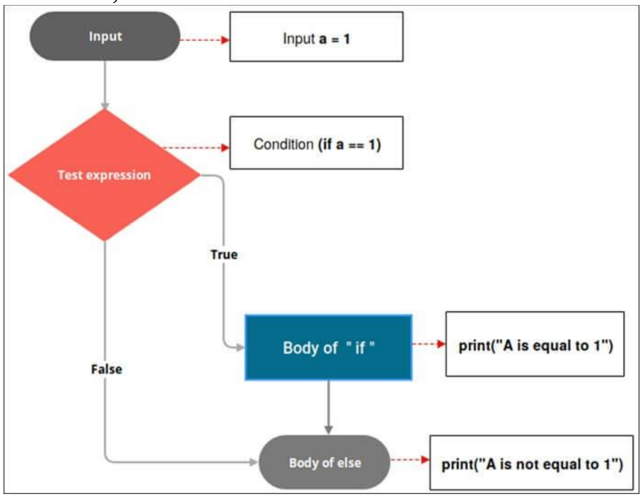
Python if. Condition

A Python if statement evaluates whether a condition is equal to true or false. The statement will execute a block of code if a specified condition is equal to true. Otherwise, the block of code within the if statement is not executed.



Python if & else Conditions

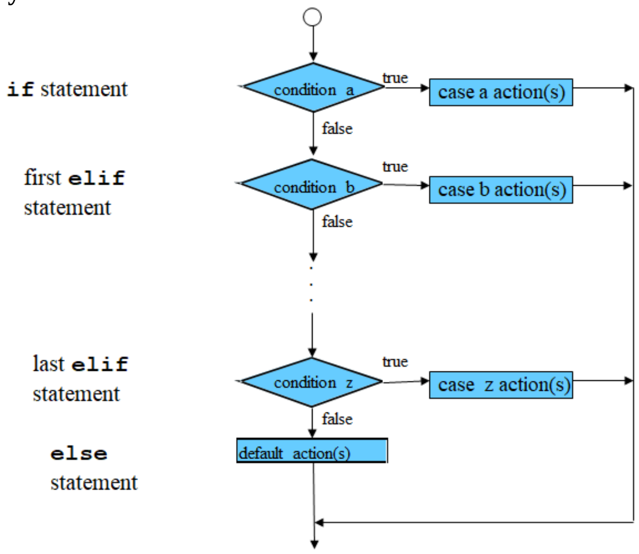
An if & else Python statement checks whether a condition is true. If a condition is true, the if statement executes. Otherwise, the else statement executes.



Python if elif & else conditions.

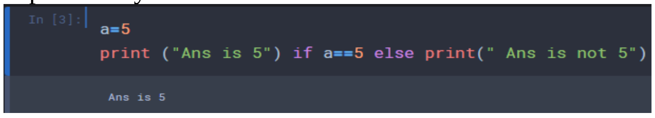
The if-elif-else statement is used to conditionally execute a statement or a block of statements. Conditions can be true or false, execute one thing when the condition is true, something else when the condition is false.

Python Decision Make If elif & else.



Python If-Else on One Line.

In Python, you can have if-else statements on one line.To write an if-else statement on one line, follow the conditional expression syntax:



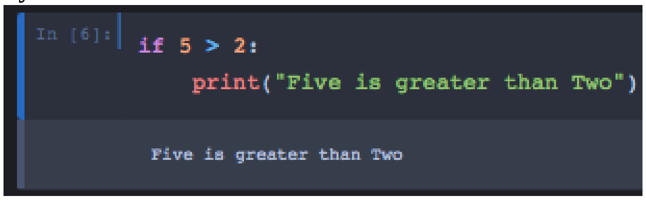
```
In [3]: a=5
        print ("Ans is 5") if a==5 else print(" Ans is not 5")

        Ans is 5
```

Our Programs:

if statement

Here is the general form of a one way if statement.

Syntax:



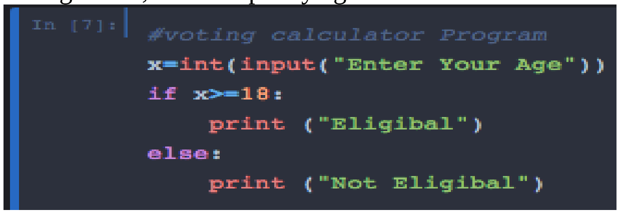
```
In [6]: if 5 > 2:
            print("Five is greater than Two")

        Five is greater than Two
```

if & else statement.

In Python if .. else statement, if has two blocks, one following the expression and other following the else clause. Here is the

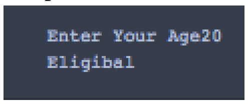
Syntax:

Write a program to check whether a person is eligible for voting or not, and accept any age from the user.

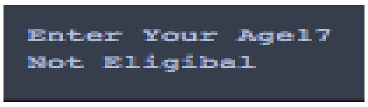


```
In [7]: #voting calculator Program
        x=int(input("Enter Your Age"))
        if x>=18:
            print ("Eligibal")
        else:
            print ("Not Eligibal")
```

Output:



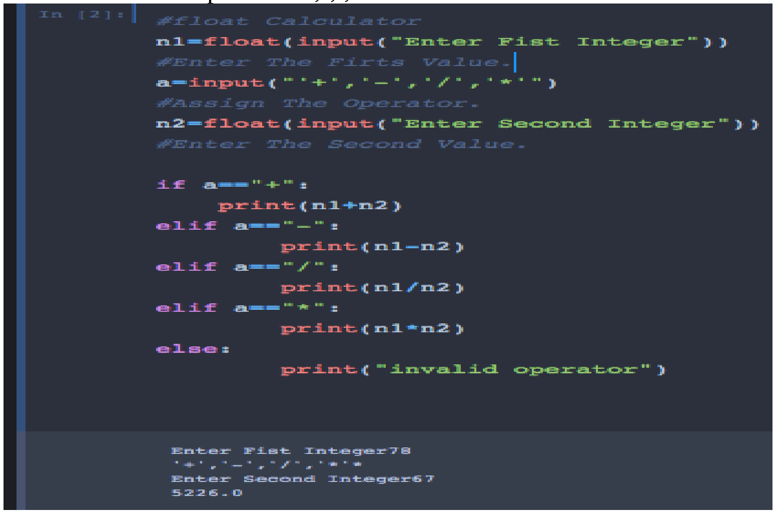
```
Enter Your Age20
Eligibal
```



```
Enter Your Age17
Not Eligibal
```

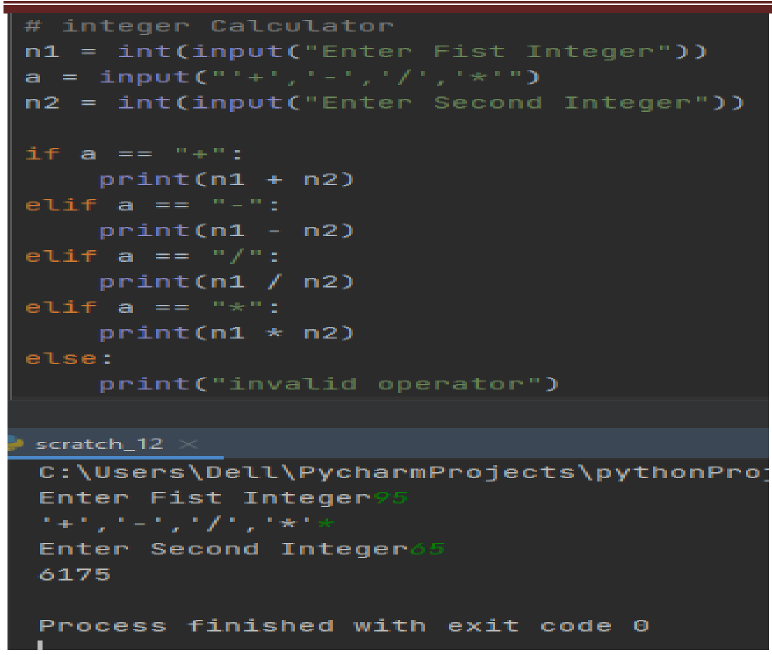
if .. elif .. else statement

Write a python code using If elif condition to make calculator that can perform +,-,/,*



```
In [2]: #float Calculator
        n1=float(input("Enter Fist Integer"))
        #Enter The Firts Value.
        a=input("'+','-','/','*'")
        #Assign The Operator.
        n2=float(input("Enter Second Integer"))
        #Enter The Second Value.

        if a=="+":
            print(n1+n2)
        elif a=="-":
                print(n1-n2)
        elif a=="/":
                print(n1/n2)
        elif a=="*":
                print(n1*n2)
        else:
                print("invalid operator")

         Enter Fist Integer78
         '+','-','/','*'*
         Enter Second Integer67
         5226.0
```

Similar Code:

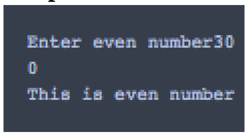


```
# integer Calculator
n1 = int(input("Enter Fist Integer"))
a = input("'+','-','/','*'")
n2 = int(input("Enter Second Integer"))

if a == "+":
    print(n1 + n2)
elif a == "-":
    print(n1 - n2)
elif a == "/":
    print(n1 / n2)
elif a == "*":
    print(n1 * n2)
else:
    print("invalid operator")
```

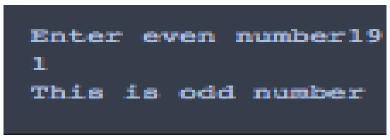
```
scratch_12
C:\Users\Dell\PycharmProjects\pythonPro
Enter Fist Integer95
'+','-','/','*'*
Enter Second Integer65
6175

Process finished with exit code 0
```

Write a program to check whether a number entered by the user is even or odd.

Output:

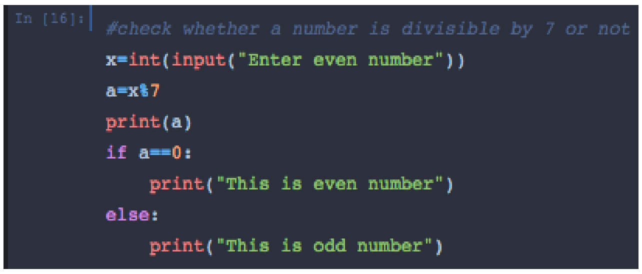


```
Enter even number30
0
This is even number
```



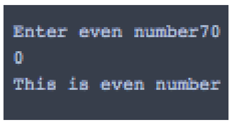
```
Enter even number19
1
This is odd number
```

Similar Code:

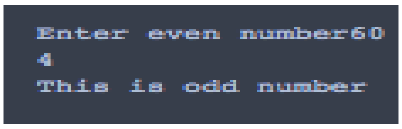


```
In [16]:  #check whether a number is divisible by 7 or not
          x=int(input("Enter even number"))
          a=x%7
          print(a)
          if a==0:
              print("This is even number")
          else:
              print("This is odd number")
```

Output:

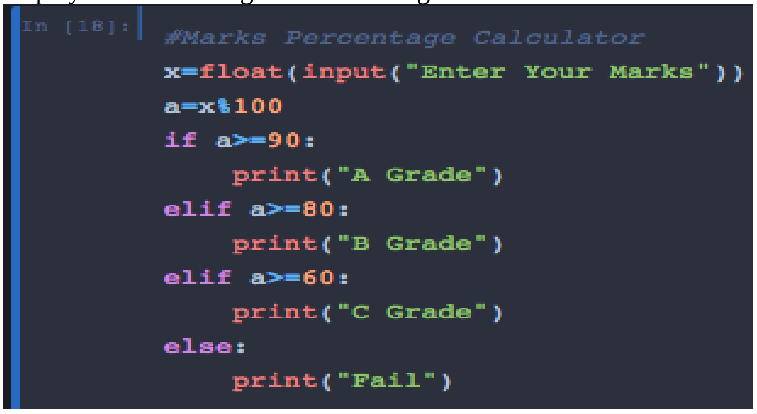


```
Enter even number70
0
This is even number
```



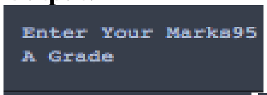
```
Enter even number60
4
This is odd number
```

Write a program to accept percentages from users and display them according to the following criteria.

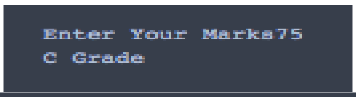


```
In [18]:  #Marks Percentage Calculator
          x=float(input("Enter Your Marks"))
          a=x%100
          if a>=90:
              print("A Grade")
          elif a>=80:
              print("B Grade")
          elif a>=60:
              print("C Grade")
          else:
              print("Fail")
```

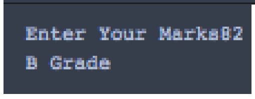
Output:



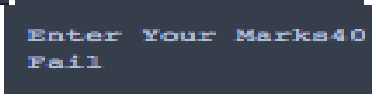
```
Enter Your Marks95
A Grade
```



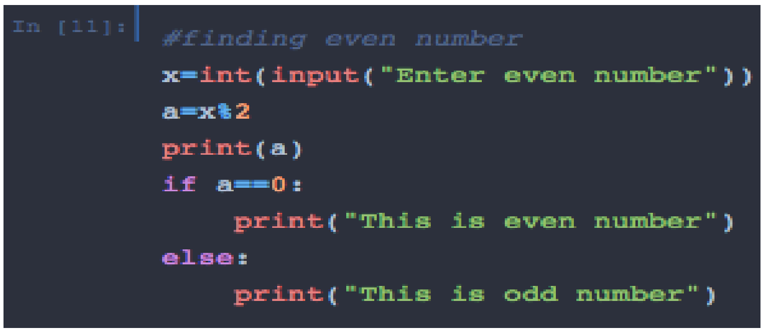
```
Enter Your Marks75
C Grade
```



```
Enter Your Marks82
B Grade
```



```
Enter Your Marks40
Fail
```



```
In [11]:  #finding even number
          x=int(input("Enter even number"))
          a=x%2
          print(a)
          if a==0:
              print("This is even number")
          else:
              print("This is odd number")
```

Introduction to LOOP in python

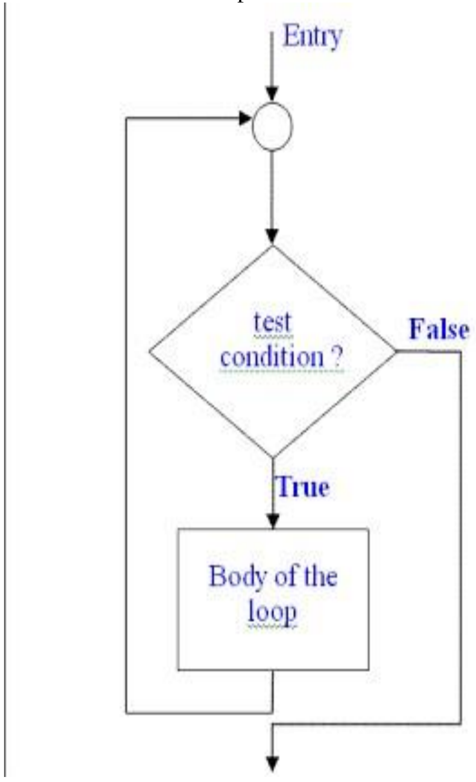
Name: Ahmad Arsim
Date of Lecture: March 2022
Date of performance: March 2022
Date of submission: May 15, 2022
Submitted to: Sir Salman Mahmood Qazi & Sir Owais Saleem
Ahmadarsim2018@gmail.com

Loop introduction

In computer Programming, a Loop is used to execute a group of instructions or a block of code multiple times, without writing it repeatedly. The block of code is executed based on a certain condition. Loops are the control structures of a program. Using Loops in computer programs simplifies rather optimizes the process of coding.

What is the structure of loop

The structure of a Loop can be virtually divided into two parts, namely the control statement, and the body. The control statement of a Loop comprises the conditions that have to be met for the execution of the body of the Loop. For every iteration of the Loop, the conditions in the control statement have to be true. The body of a Loop comprises the block of code or the sequence of logical statements that are to be executed multiple times.



Types of loops

There are two types of Loops in Python, namely,

1. While Loop.
2. For Loop

Nested loop

When a Loop is written within another Loop, the control structure is termed as a nested Loop.

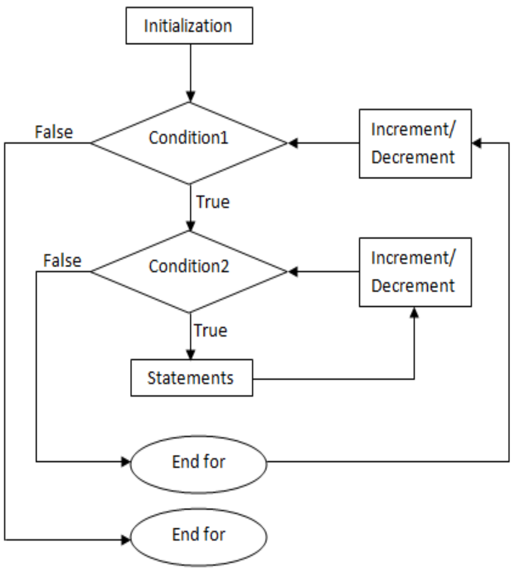
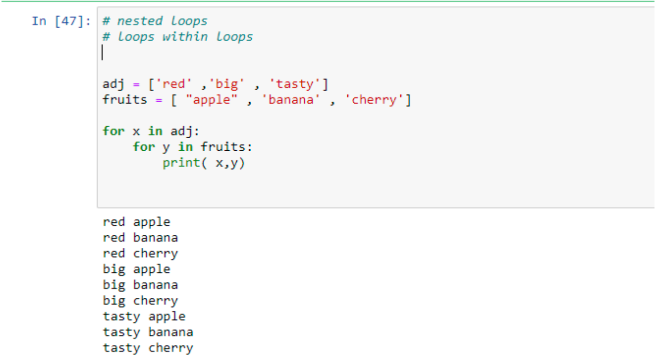


Fig: Flowchart for nested for loop



```
In [47]: # nested loops
         # loops within loops
         |

         adj = ['red' ,'big' , 'tasty']
         fruits = [ "apple" , 'banana' , 'cherry']

         for x in adj:
             for y in fruits:
                 print( x,y)
```

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

While Loop

In most computer programming languages, a while loop is a control flow statement that allows code to be executed repeatedly. The while loop can be thought of as a repeating if statement. The while construct consists of a block of code and a condition. The condition is evaluated, and if the condition is true, the code within the block is executed. This repeats until the condition becomes false. Because while loop checks the condition before the block is executed, the control structure is often also known as a pre-test.

The syntax of a while loop in Python programming language is

Syntax

while expression:
 statement(s)

Here, statement(s) may be a single statement or a block of statements.

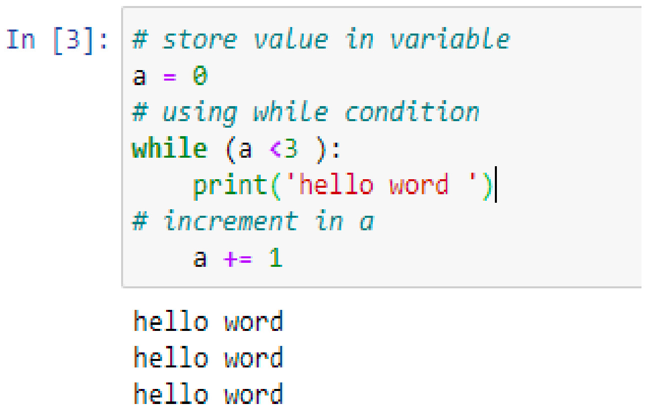
Q) Print Hello three time

# store value in variable

```
a = 0
# using while condition
while (a <3):
    print('hello word ')
# increment in a
    a += 1
```

OUTPUT

```
hello word
hello word
hello word
```



```
In [3]: # store value in variable
        a = 0
        # using while condition
        while (a <3 ):
            print('hello word ')
        # increment in a
            a += 1

        hello word
        hello word
        hello word
```

Q) print the series of number from 1 to 6

```
# store value in variable
i = 1
# using while condition
while i <6:
    print(i)
# increment in i
    i +=1
```

OUTPUT

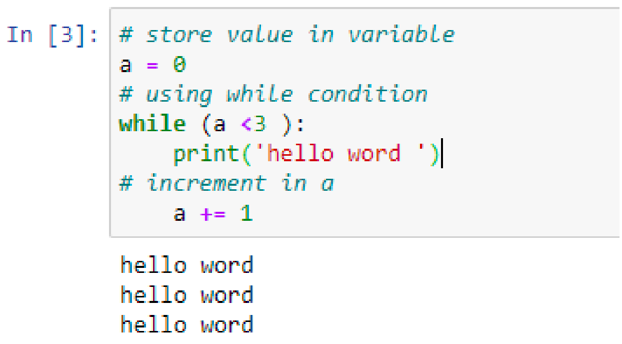
```
1
2
3
4
5
```

Q) print the Square of the number

```
num = 1
print("Numbers Squares")
while(num<=10):
# t givs tab space
   print(num,"\t", num ** 2)
   num = num + 1
```

OUTPUT

```
Numbers Squares
1         1
2         4
3         9
4         16
5         25
6         36
7         49
8         64
9         81
10        100
```



```
In [3]: # store value in variable
        a = 0
        # using while condition
        while (a <3 ):
            print('hello word ')
        # increment in a
            a += 1

        hello word
        hello word
        hello word
```

Q) print the reverse number ?

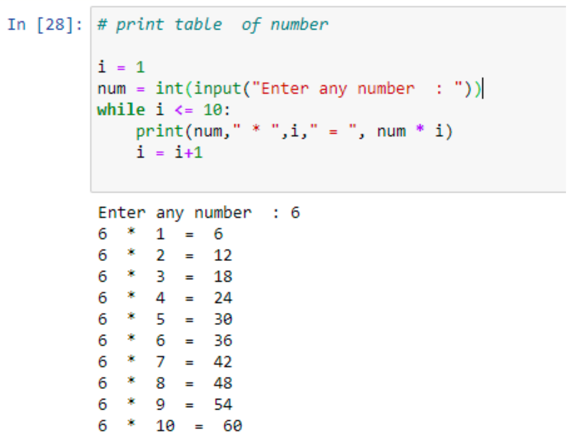
```
num = 10
while num >= 1:
    print(num)
    num= num - 1
# this will reverse the number
```

OUTPUT

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

Q) print table of number

```
i = 1
num = int(input("Enter any number  : "))
while i <= 10:
   print(num," * ",i," = ", num * i)
   i = i+1
```



```
In [28]: # print table  of number

         i = 1
         num = int(input("Enter any number  : "))
         while i <= 10:
             print(num," * ",i," = ", num * i)
             i = i+1

         Enter any number  : 6
         6  *  1  =  6
         6  *  2  =  12
         6  *  3  =  18
         6  *  4  =  24
         6  *  5  =  30
         6  *  6  =  36
         6  *  7  =  42
         6  *  8  =  48
         6  *  9  =  54
         6  *  10  =  60
```

USING BREAK CONDITION

Q) Using break condition to break the program on 6?

```
i = 1
# using while condition
while i < 10:
   print (i)
# here we use a break condition
# the break condition breaks the program where the required condition meets
   if i == 6:
      break
```

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

USING CONTINUE STATEMENT

Q) Using Continue statement to stop at 5 and then continue to print till 10

```
i = 2
while i < 10:
   i +=1
   if i == 5:
       continue
   print ( i)
```

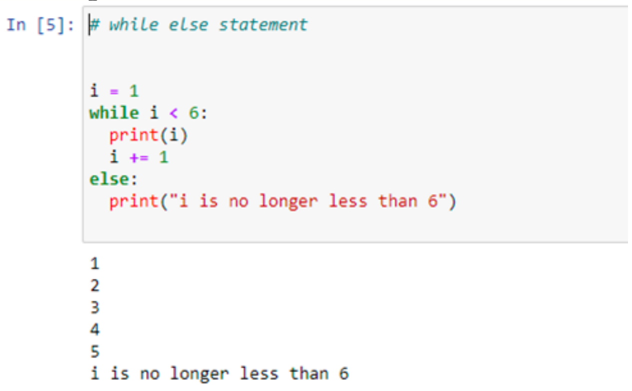
# With the continue statement we can stop the current iteration, and continue with the next

```
 OUTPUT
3
4
6
7
8
9
10
```

WITH ELSE STATEMENT

Q) print series from 1 to 5 using while condition

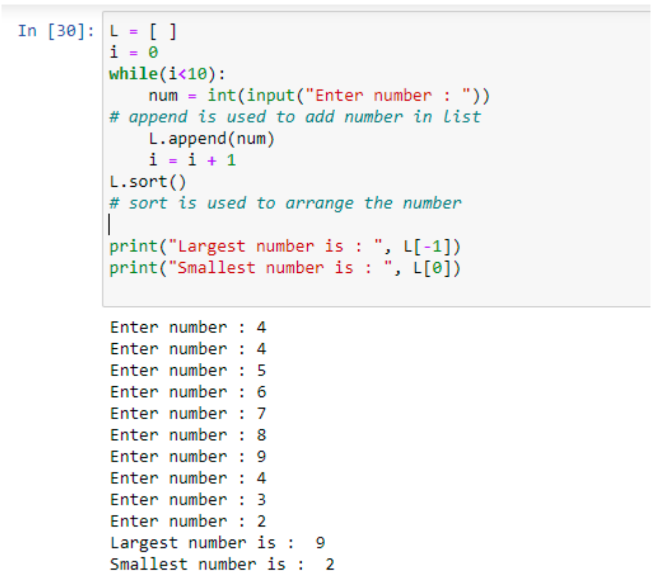
```
i = 1
while I <6:
  print(i)
  i +=1
  else:
      Print("I is no longer than 6")
# print both condition
OUTPUT
1
2
3
4
5
i is no longer less than 6
```



```
In [5]: # while else statement


        i = 1
        while i < 6:
          print(i)
          i += 1
        else:
          print("i is no longer less than 6")

1
2
3
4
5
i is no longer less than 6
```

Q) **Print the greatest and smaller number?**

```
L = [ ]
i = 0
while(i<10):
    num = int(input("Enter number : "))
    L.append(num)
# append is used to add number in list
    i = i + 1
L.sort()
# sort is used to arrange the number
          print("Largest number is : ", L[-1])
          print("Smallest number is : ", L[0])
     OUTPUT
Enter number : 4
Enter number : 4
Enter number : 5
Enter number : 6
Enter number : 7
Enter number : 8
Enter number : 9
Enter number : 4
Enter number : 3
Enter number : 2
Largest number is :  9
Smallest number is :  2
```

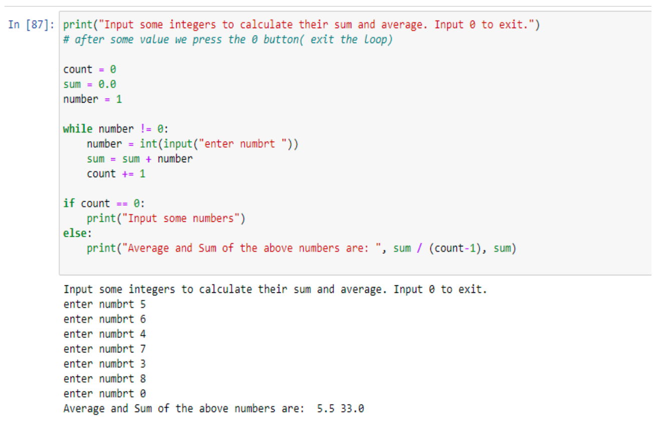


```
In [30]: L = [ ]
         i = 0
         while(i<10):
             num = int(input("Enter number : "))
         # append is used to add number in list
             L.append(num)
             i = i + 1
         L.sort()
         # sort is used to arrange the number

         print("Largest number is : ", L[-1])
         print("Smallest number is : ", L[0])

         Enter number : 4
         Enter number : 4
         Enter number : 5
         Enter number : 6
         Enter number : 7
         Enter number : 8
         Enter number : 9
         Enter number : 4
         Enter number : 3
         Enter number : 2
         Largest number is :  9
         Smallest number is :  2
```

Q) Find the average number?

print("Input some integers to calculate their sum and average. Input 0 to exit.")

```
# after some value we press the 0 button( exit the loop)
count = 0
sum = 0.0
number = 1
while number != 0:
    number = int(input("enter numbrt "))
    sum = sum + number
    count += 1
if count == 0:
    print("Input some numbers")
else:
    print("Average and Sum of the above numbers are: ", sum / (count-1), sum)
OUTPUT
Input some integers to calculate their sum and average.
Input 0 to exit.
enter numbrt 5
enter numbrt 6
enter numbrt 4
enter numbrt 7
enter numbrt 3
enter numbrt 8
enter numbrt 0
Average and Sum of the above numbers are:  5.5 33.0
```



```
In [87]: print("Input some integers to calculate their sum and average. Input 0 to exit.")
         # after some value we press the 0 button( exit the loop)

         count = 0
         sum = 0.0
         number = 1

         while number != 0:
             number = int(input("enter numbrt "))
             sum = sum + number
             count += 1

         if count == 0:
             print("Input some numbers")
         else:
             print("Average and Sum of the above numbers are: ", sum / (count-1), sum)

         Input some integers to calculate their sum and average. Input 0 to exit.
         enter numbrt 5
         enter numbrt 6
         enter numbrt 4
         enter numbrt 7
         enter numbrt 3
         enter numbrt 8
         enter numbrt 0
         Average and Sum of the above numbers are:  5.5 33.0
```

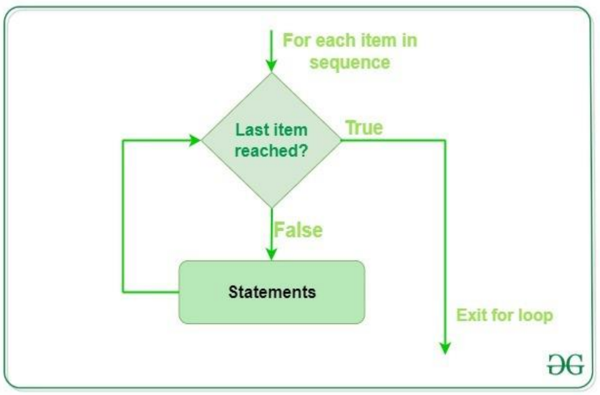
FOR LOOP

A for loop in Python is a control flow statement that is used to repeatedly execute a group of statements as long as the condition is satisfied. Such a type of statement is also known as an iterative statement. Therefore, a for loop is an iterative statement.

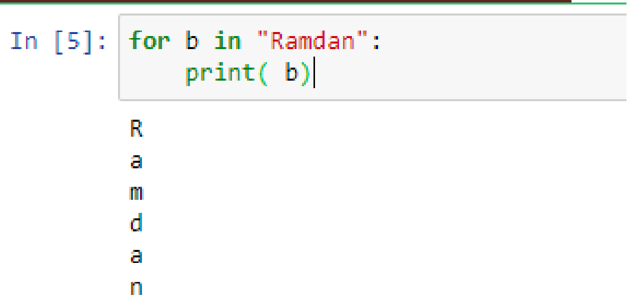
The statements in any Python program are always executed from top to bottom. However, you can control the flow of execution by employing control flow statements such as a for loop. Usually, the for loop is used when we know in advance how many times a code block needs to be executed.

Syntax:

```
for var in iterable:
          # statements
```



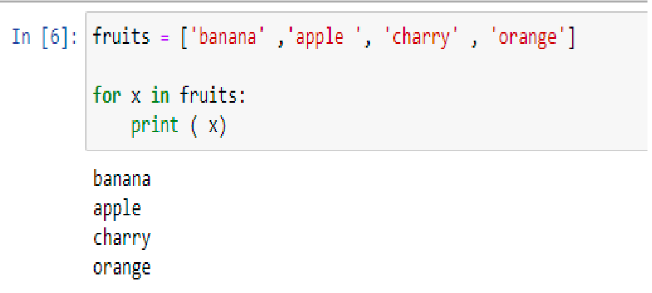
```
for b in "Ramdan":
   print( b)
OUTPUT
      R
      a
      m
      d
      a
      n
```



```
In [5]: for b in "Ramdan":
            print( b)

        R
        a
        m
        d
        a
        n
```

Q) Print the fruits name using for loop?

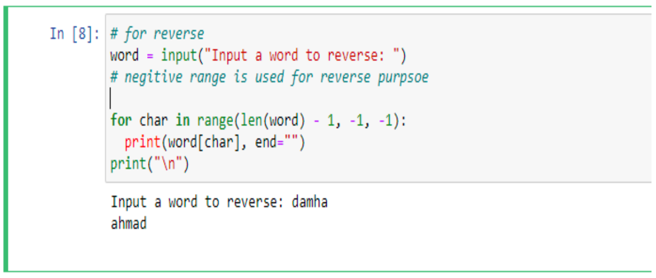
```
fruits = ['banana' ,'apple ', 'charry' , 'orange']
for x in fruits:
   print ( x)
Output
banana
apple
charry
orange
```



```
In [6]: fruits = ['banana' ,'apple ', 'charry' , 'orange']

        for x in fruits:
            print ( x)

        banana
        apple
        charry
        orange
```

Q) Find reverse Number using For loop?

```
word = input("Input a word to reverse: ")
# negitive range is used for reverse purpsoe
for char in range(len(word) - 1, -1, -1):
  print(word[char], end="")
print("\n")
OUTPUT
Input a word to reverse: damha
ahmad
```



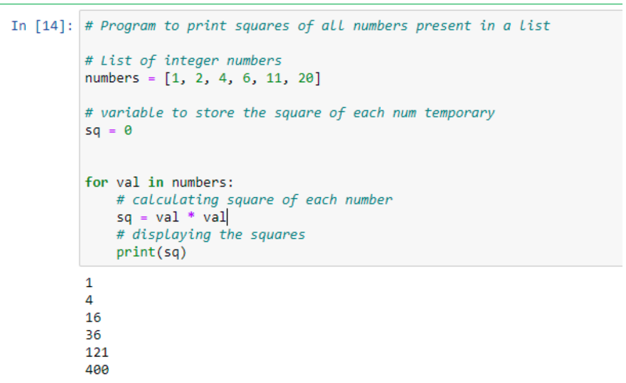
```
In [8]: # for reverse
        word = input("Input a word to reverse: ")
        # negitive range is used for reverse purpsoe

        for char in range(len(word) - 1, -1, -1):
          print(word[char], end="")
        print("\n")

        Input a word to reverse: damha
        ahmad
```

# Program to print squares of all numbers present in a list

Q) find the square of the given list?

```
numbers = [1, 2, 4, 6, 11, 20]
# variable to store the square of each value
sq = 0
for val in numbers:
   # calculating square of each number
   sq = val * val
   # displaying the squares
   print(sq)
OUTPUT
```

```
1
4
16
36
121
400
```



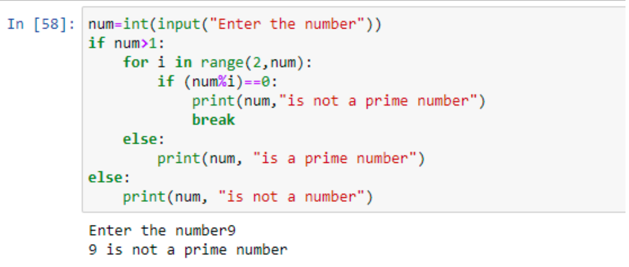
```
In [14]: # Program to print squares of all numbers present in a list

         # List of integer numbers
         numbers = [1, 2, 4, 6, 11, 20]

         # variable to store the square of each num temporary
         sq = 0


         for val in numbers:
             # calculating square of each number
             sq = val * val
             # displaying the squares
             print(sq)

         1
         4
         16
         36
         121
         400
```

Q) Print the Prime number?

```
num=int(input("Enter the number"))
if num>1:
    for i in range(2,num):
        if (num%i)==0:
            print(num,"is not a prime number")
            break
    else:
        print(num, "is a prime number")
else:
    print(num, "is not a number")
```

OUTPUT

```
Enter the number9
9 is not a prime number
```



```
In [58]: num=int(input("Enter the number"))
         if num>1:
             for i in range(2,num):
                 if (num%i)==0:
                     print(num,"is not a prime number")
                     break
             else:
                 print(num, "is a prime number")
         else:
             print(num, "is not a number")

         Enter the number9
         9 is not a prime number
```

USING BREAK FUNCTION

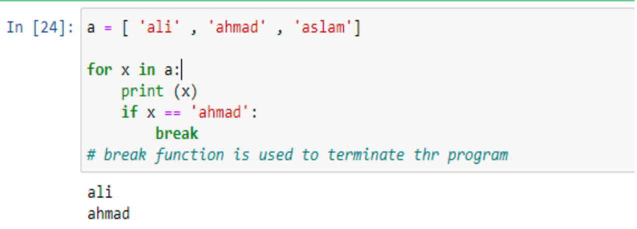
Q) break the function using brake function?

a = ['ali' , 'ahmad' , 'aslam']

```
for x in a:

    print (x)

    if x == 'ahmad':

        break

# break function is used to terminate thr program
```

OUTPUT

```
        ali
        ahmad
```



```
In [24]: a = [ 'ali' , 'ahmad' , 'aslam']

         for x in a:
             print (x)
             if x == 'ahmad':
                 break
         # break function is used to terminate thr program

         ali
         ahmad
```

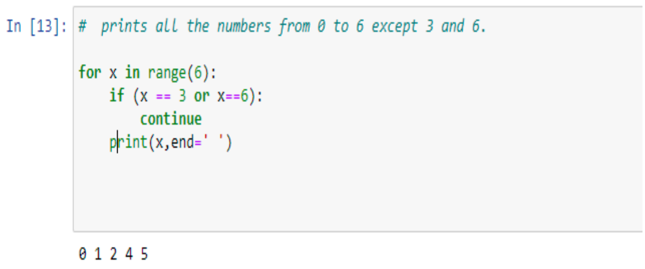
CONTINUE STATEMENT

Q) prints all the numbers from 0 to 6 except 3 and 6.

```
for x in range(6):
    if (x == 3 or x==6):
        continue
    print(x,end=' ')
```

OUTPUT

```
0 1 2 4 5
```



```
In [13]: #  prints all the numbers from 0 to 6 except 3 and 6.

         for x in range(6):
             if (x == 3 or x==6):
                 continue
             print(x,end=' ')



         0 1 2 4 5
```

RANGE

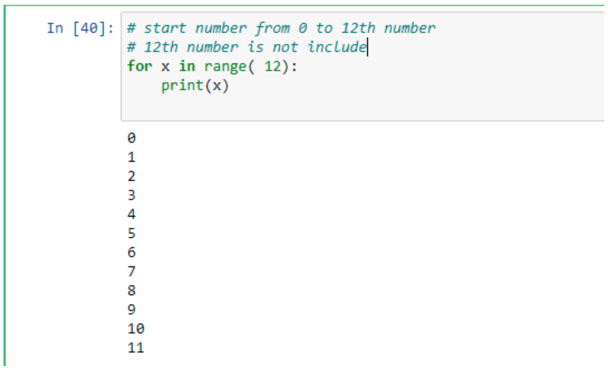
Q) Print series using range function?

range(start, stop ,[step(increment)])

```
# start number from 0 to 12th number
# 12th number is not include

for x in range( 12):
    print(x)
```

OUTPUT

```
0
1
2
3
4
5
6
7
8
9
10
11
```



```
In [40]: # start number from 0 to 12th number
         # 12th number is not include
         for x in range( 12):
             print(x)

         0
         1
         2
         3
         4
         5
         6
         7
         8
         9
         10
         11
```

Q) print the range start from 2 to 10 with the increment of 2?

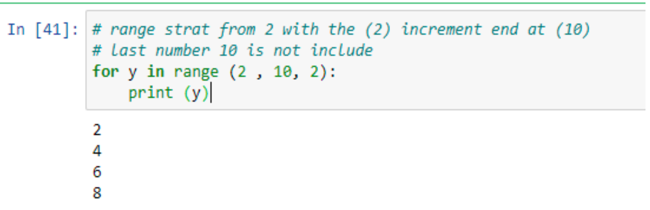
# range strat from 2 with the (2) increment end at (10)

last number 10 is not include

```
for y in range (2 , 10, 2):
    print (y)
```

output

```
2
4
6
8
```



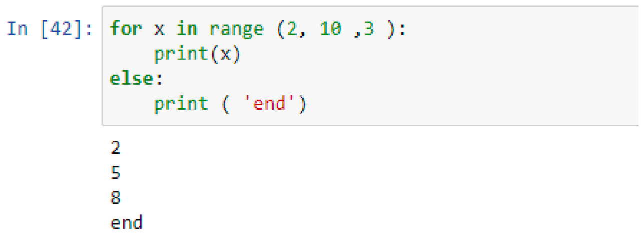
```
In [41]: # range strat from 2 with the (2) increment end at (10)
         # last number 10 is not include
         for y in range (2 , 10, 2):
             print (y)

         2
         4
         6
         8
```

Q) print the range start from 2 to 10 with the increment of 3?

```
for x in range (2, 10 ,3 ):
    print(x)
else:
    print ( 'end')
```

OUTPUT

```
2
5
8
end
```

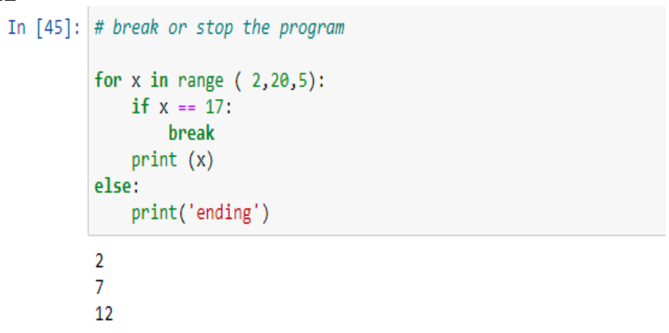


```
In [42]: for x in range (2, 10 ,3 ):
             print(x)
         else:
             print ( 'end')

         2
         5
         8
         end
```

```
# break or stop the program
for x in range ( 2,20,5):
    if x == 17:
        break
    print (x)
else:
    print('ending')
```

OUTPUT

```
2
7
12
```



```
In [45]: # break or stop the program

         for x in range ( 2,20,5):
             if x == 17:
                 break
             print (x)
         else:
             print('ending')

         2
         7
         12
```

NESTED LOOP

nested loops

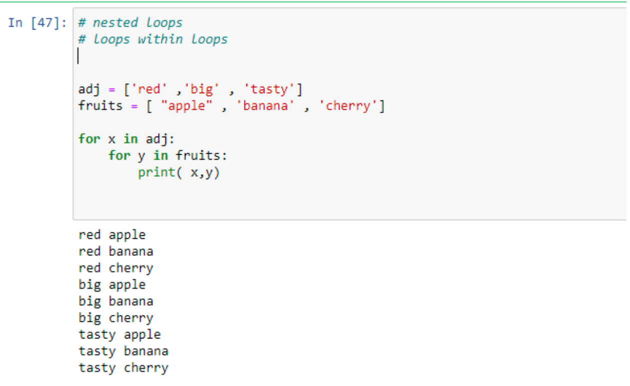
loops within loops

Q) Loop within loop?

```
adj = ['red' ,'big' , 'tasty']
fruits = [ "apple" , 'banana' , 'cherry']

for x in adj:
    for y in fruits:
        print( x,y)
```

OUTPUT

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```



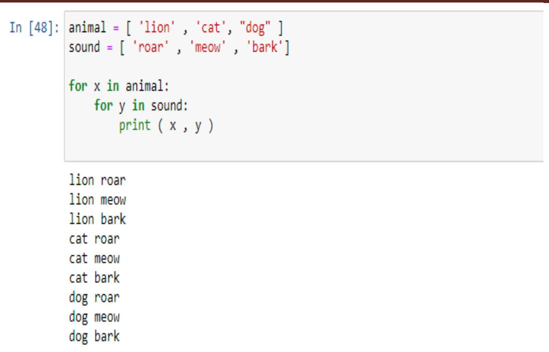
```
In [47]: # nested loops
         # loops within loops

         adj = ['red' ,'big' , 'tasty']
         fruits = [ "apple" , 'banana' , 'cherry']

         for x in adj:
             for y in fruits:
                 print( x,y)

         red apple
         red banana
         red cherry
         big apple
         big banana
         big cherry
         tasty apple
         tasty banana
         tasty cherry
```

Q) Loop within loop?

```
animal = [ 'lion' , 'cat', "dog" ]
sound = [ 'roar' , 'meow' , 'bark']

for x in animal:
    for y in sound:
        print ( x , y )
```

OUTPUT

```
lion roar
lion meow
lion bark
cat roar
cat meow
cat bark
dog roar
dog meow
dog bark
lion meow
lion bark
cat roar
cat meow
cat bark
dog roar
dog meow
dog bark
```



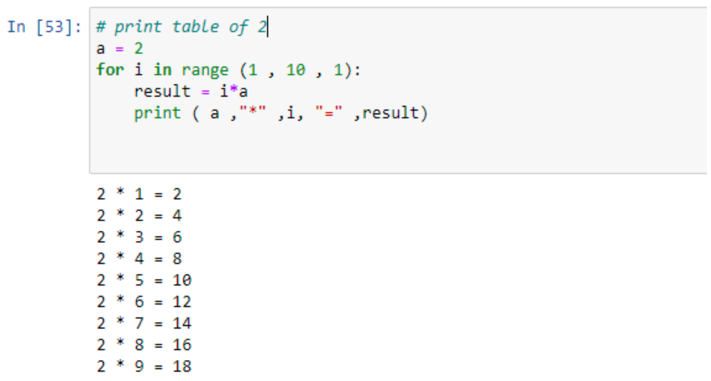
```
In [48]: animal = [ 'lion' , 'cat', "dog" ]
         sound = [ 'roar' , 'meow' , 'bark']

         for x in animal:
             for y in sound:
                 print ( x , y )
```

```
lion roar
lion meow
lion bark
cat roar
cat meow
cat bark
dog roar
dog meow
dog bark
```

Q) Print the table using for loop?

```
# table of 2
# print table of 2
a = 2
for i in range (1 , 11 , 1):
    result = i*a
    print ( a ,"*" ,i, "=" ,result)
OUTPUT
```

2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
2 * 10= 20



```
In [53]: # print table of 2
         a = 2
         for i in range (1 , 10 , 1):
             result = i*a
             print ( a ,"*" ,i, "=" ,result)
```

```
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
```

Introduction to Functions in Python

Name: Muhammad Habib
Date of Lecture: March 2022
Date of performance: March 2022
Date of submission: May 16, 2022
Submitted to: Sir Salman Mahmood Qazi & Sir Owais Saleem
mh.habib010@gmail.com

Introduction

In programming, A function is a block of code or a group of statements that performs a particular task .

Functions are the most important aspect of an application. A function can be defined as the organized block of reusable code, which can be called whenever required.

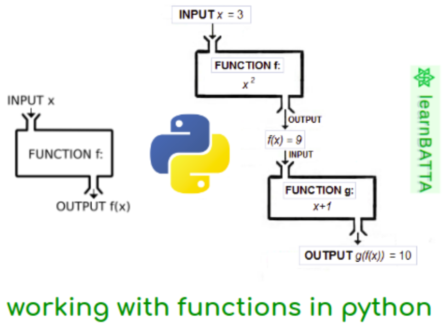
Python allows us to divide a large program into the basic building blocks known as a function. The function contains the set of programming statements in the body of function. A function can be called multiple times to provide reusability to the Python program.

Functions can be both built-in or user-defined.

- **Built-in functions** - The built-in functions are those functions that are **pre-defined** in Python. i.g. range(), print(), type(), abs(), bin() etc.

- **User-define functions** - The user-defined functions are those define by the **user** to perform the specific task.

Advantage of Function In Python



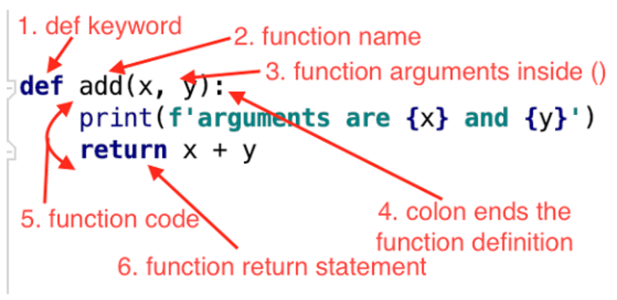
There are the following advantages of Python functions.

- Using functions, we can avoid rewriting the same logic/code again and again in a program.

- We can call Python functions multiple times in a program and anywhere in a program.

- We can track a large Python program easily when it is divided into multiple functions.

- Reusability is the main achievement of Python functions.
- However, Function calling is always overhead in a Python program.



```
def add(x, y):
    print(f'arguments are {x} and {y}')
    return x + y
```

1. def keyword
2. function name
3. function arguments inside ()
4. colon ends the function definition
5. function code
6. function return statement

Creating a Function

We can create a Python function using by **def** keyword.

To create a function, we need the following:

- The **def** keyword.
- A function name.
- Round brackets () and a semicolon :
- A function body - a group of statements.

Syntax:https://www.javatpoint.com/python-functionshttps://www.javatpoint.com/python-functionshttps://www.javatpoint.com/python-functions

1. **def** my_function(parameters):
2. function_block
3. **return** expression

Example:

```
def my_function():
      print("Hello from a function")
```

Note! A group of statements must have the same indentation level, in the example above we used 3 whitespaces to indent the function body.

Calling a Function

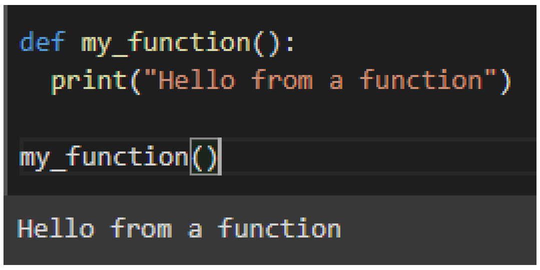
To call a function, use the function name followed by parenthesis:

Examples:

```
def my_function():
      print("Hello from a function")
my_function()
```

Output:

Hello from a function



```
def my_function():
  print("Hello from a function")

my_function()
```

```
Hello from a function
```

Function Name

A function name to uniquely identify the function. Here we define a function with the name of "hello".

def hello ():

Function Parameters

A parameter is a variable declared in the function. In this example, the hello() function has one parameter, the name parameter.

```
def hello(name):
      x = "Hello"
      print(x)
```

Function Arguments

An argument is the value passed to the function when it is called. You can add as many arguments as you want, just separate them with a comma.

```
def hello(name):
      x = "Hello"
      print(x)
hello("John")
```

In this example, we are calling the hello() function with one argument, "John".

Multiple Parameters/Arguments

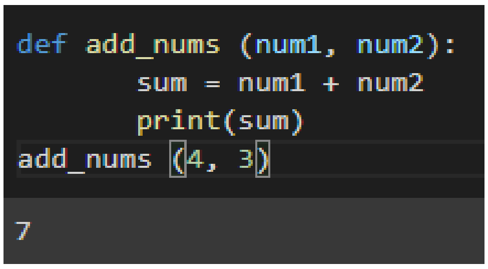
To use multiple parameters, separate them using commas.

The arguments when calling the function should also be separated by commas.

```
def add_nums(num1, num2):
      sum = num1 + num2
      print(sum)
add_nums (4, 3)
```

Output:

7

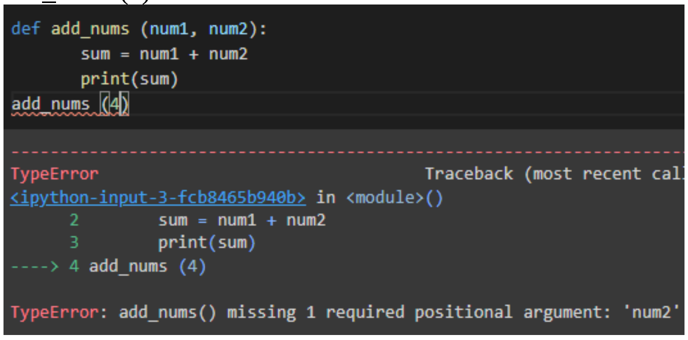


```
def add_nums (num1, num2):
        sum = num1 + num2
        print(sum)
add_nums (4, 3)
```

```
7
```

Note! You will get an error if you don't pass enough required arguments.

```
def add_nums (num1,num2):
     sum = num1 + num2
     print (sum)
# only one argument is passed    and    # 2 are arguments are required
add_nums (4)
```



```
def add_nums (num1, num2):
        sum = num1 + num2
        print(sum)
add_nums (4)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent cal
<ipython-input-3-fcb8465b940b> in <module>()
      2         sum = num1 + num2
      3         print(sum)
----> 4 add_nums (4)

TypeError: add_nums() missing 1 required positional argument: 'num2'
```

Default Arguments

A function can have default arguments.

It can be done using the assignment operator (=).

If you don't pass the argument, the default argument will be used instead.

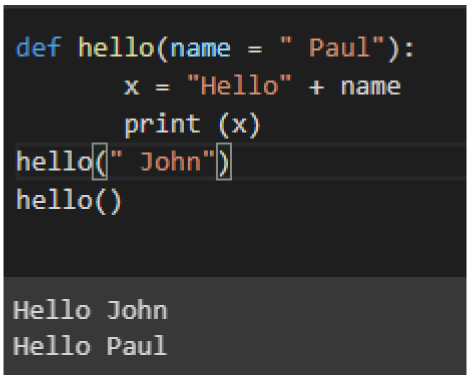
Example:

def hello(name = " Paul"):

```
    x = "Hello" + name
    print (x)
hello(" John")
hello()
```

Output:

```
Hello John
Hello Paul
```



```
def hello(name = " Paul"):
        x = "Hello" + name
        print (x)
hello(" John")
hello()

Hello John
Hello Paul
```

Keyword Arguments

As you may notice, the arguments passed are assigned according to their postion.

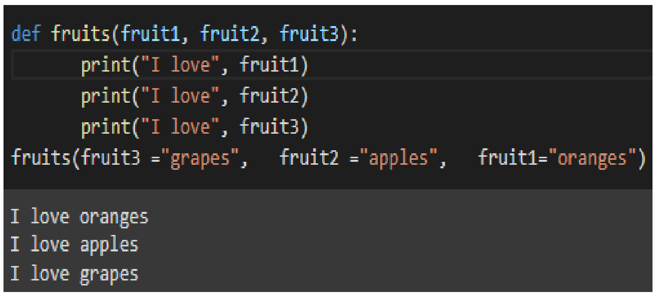
When using keyword arguments, the position does not matter.

Example:

```
def fruits(fruit1, fruit2, fruit3):
    print("I love", fruit1)
    print("I love", fruit2)
    print("I love", fruit3)

fruits(fruit3 ="grapes",   fruit2 ="apples",   fruit1="oranges")
```

Output:

```
I love oranges
I love apples
I love grapes
```



```
def fruits(fruit1, fruit2, fruit3):
        print("I love", fruit1)
        print("I love", fruit2)
        print("I love", fruit3)
fruits(fruit3 ="grapes",   fruit2 ="apples",   fruit1="oranges")

I love oranges
I love apples
I love grapes
```

Arbitrary Arguments, *args

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:

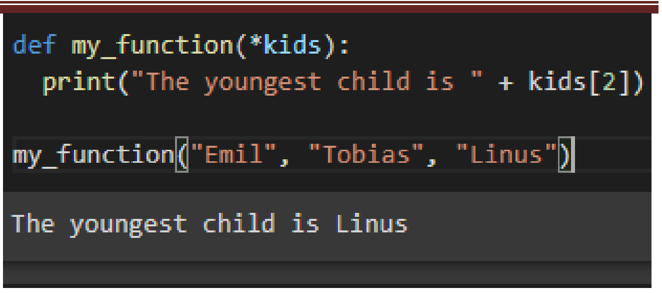
Example:

If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])
my_function("Emil", "Tobias", "Linus")
```

Output:

```
The youngest child is Linus
```



```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")

The youngest child is Linus
```

Arbitrary Keyword Arguments, **kwargs

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:

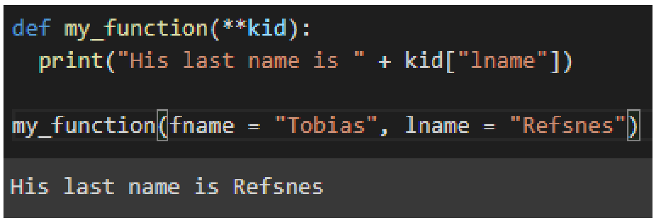
Example:

If the number of keyword arguments is unknown, add a double ** before the parameter name:

```
def my_function(**kid):
  print("His last name is " + kid["lname"])
my_function(fname = "Tobias", lname = "Refsnes")
```

Output:

```
His last name is Refsnes
```



```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")

His last name is Refsnes
```

The Return Statement

The return statement is used at the end of the function and returns the result of the function. It terminates the function execution and transfers the result where the function is called. The return statement cannot be used outside of the function.

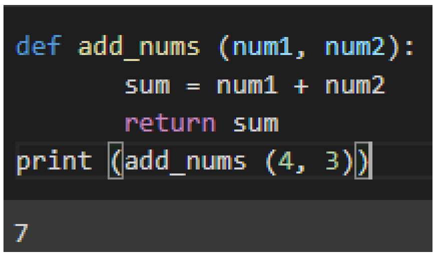
Syntax: return [expression_list]

Example:

```
def add_nums (num1, num2):
    sum = num1 + num2
    return sum
print (add_nums (4, 3))
```

Output:

```
7
```



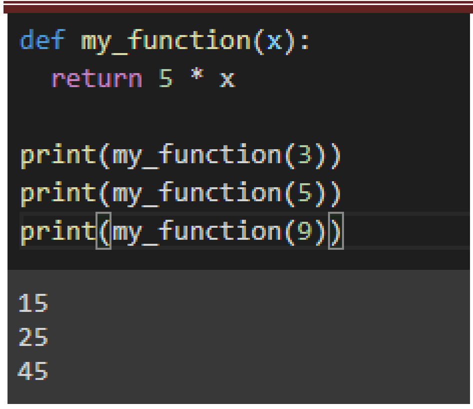
```
def add_nums (num1, num2):
        sum = num1 + num2
        return sum
print (add_nums (4, 3))

7
```

Example:

```
def my_function(x):
  return 5 * x
print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Output:

```
15
25
45
```



```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

```
15
25
45
```

The pass Statement

Function definitions cannot be empty, but for some reason if you have a function definition with no content, put in the pass statement to avoid getting an error.

Example:

```
def myfunction():
  pass
```

Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

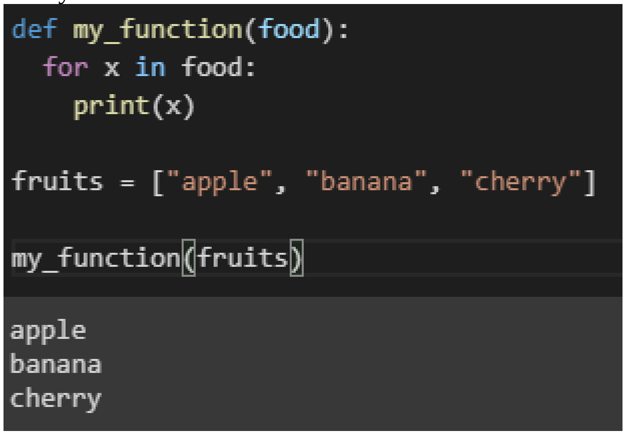
E.g. if you send a List as an argument, it will still be a List when it reaches the function:

Example:

```
def my_function(food):
  for x in food:
    print(x)
fruits=["apple", "banana", "cherry"]
my_function(fruits)
```

Output:

```
apple
banana
cherry
```



```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

```
apple
banana
cherry
```

Recursion

Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

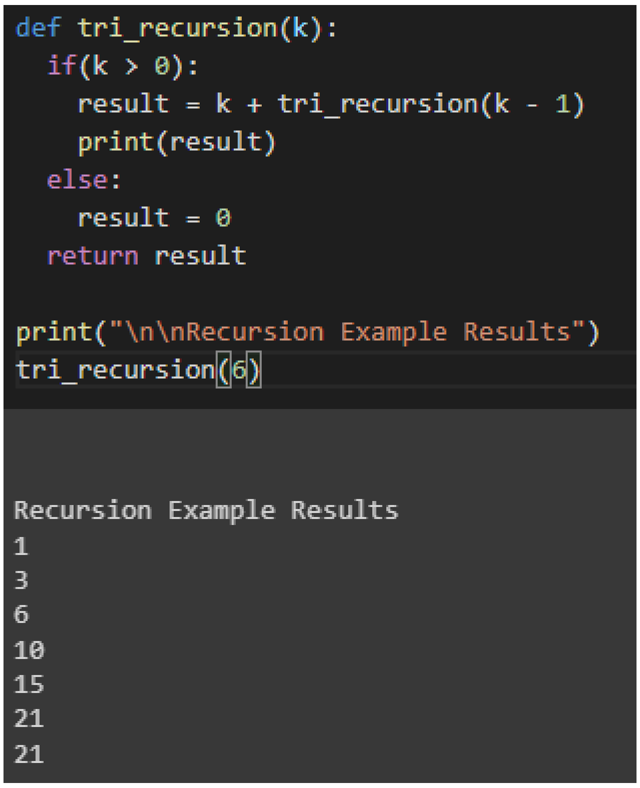
Example:

In this example, tri_recursion() is a function that we have defined to call itself ("recurse"). We use the k variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

```
def  tri_recursion(k):
# Function defined "tri_recursion" with one parameter"k".
 if(k > 0):
# if condition applied to check that passed argument as "k" is greater than 0 or not.
   result = k + tri_recursion(k - 1)
# here k + tri_recursion (k - 1) store in new variable "result"
 print(result)
#print the result.
 else:
   result = 0
# 0 in result if k not greater than 0.
return result
# return statement use to return result.
print("\n\nRecursion  Example  Results")
tri_recursion(6)
# function calling.
```

Output:

```
Recursion Example Results
1
3
6
10
15
21
21
```



```
def tri_recursion(k):
  if(k > 0):
    result = k + tri_recursion(k - 1)
    print(result)
  else:
    result = 0
  return result

print("\n\nRecursion Example Results")
tri_recursion(6)
```

```


Recursion Example Results
1
3
6
10
15
21
21
```

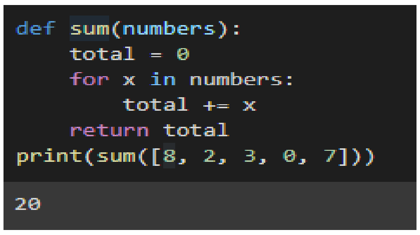
PROGRAMS USING FUNCTIONS

1. Write a Python function to sum all the numbers in a list.

```
# by defining a function.
def  sum(numbers):
# Function defined "sum" with one parameter "numbers".
   total = 0
 # counter setting.
   for x in numbers:
        # for loop used to access the elements of "numbers" and
store in "x" respectively
            total += x
            # increment of x in total with respectively.
        return total
    print(sum([8, 2, 3, 0, 7]))
    # NOW print the function.
```

Output:

20



```
def sum(numbers):
    total = 0
    for x in numbers:
        total += x
    return total
print(sum([8, 2, 3, 0, 7]))
```

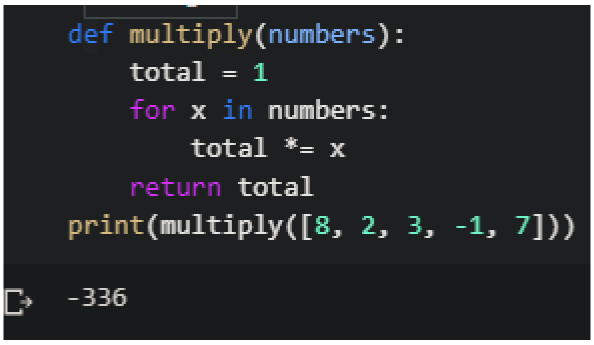
```
20
```

2. Write a Python function to multiply all the numbers in a list.

```
    # by defining a function..
    def multiply(numbers):
    # Function defined "multiply" with one parameter
"numbers".
        total = 1
        # counter setting.
        for x in numbers:
        # for loop used to access the elements of "numbers" and
store in "x" respectively.
            total *= x
          # multier increment of "x" in "total" with respectively.
        return total
    print(multiply([8, 2, 3, -1, 7]))
    # NOW print the function by giving value '5'.
```

Output:

-336



```
def multiply(numbers):
    total = 1
    for x in numbers:
        total *= x
    return total
print(multiply([8, 2, 3, -1, 7]))
```

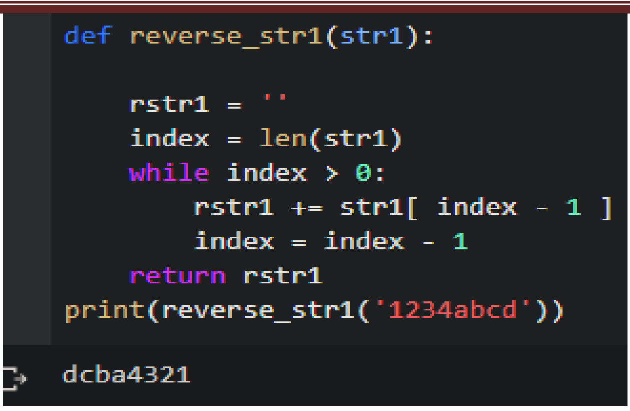
```
-336
```

3. Write a Python program to reverse a string.

```
    # by defining a function..
    def  reverse_str1(str1):
    # Function defined "test_prime" with one parameter "n".
        rstr1 = ''
        index = len(str1)
        while index > 0:
            rstr1 += str1[ index - 1 ]
            index = index - 1
        return rstr1
    print(reverse_str1('1234abcd'))
    # NOW print the function by giving string.
```

Output:

dcba4321



```
def reverse_str1(str1):

    rstr1 = ''
    index = len(str1)
    while index > 0:
        rstr1 += str1[ index - 1 ]
        index = index - 1
    return rstr1
print(reverse_str1('1234abcd'))
```

```
dcba4321
```

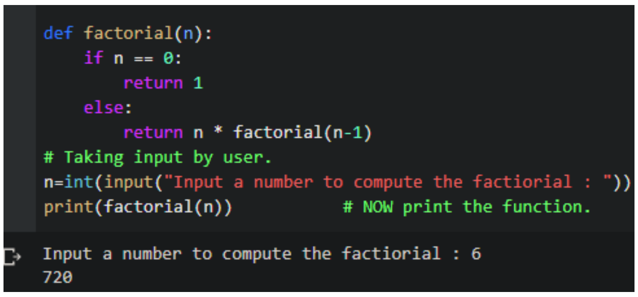
4. Write a Python function to calculate the factorial of a number (a non-negative integer).

```
    # by defining a function..
    def factorial(n):
    # Function defined "factorial" with one parameter "n".
        if  n == 0:
            return 1
    # condition applied that if passed argument n is 0 then
return 1, otherwise return 'n * (n-1)'s factorial'.
        else:
            return n * factorial(n-1)
    # Taking input by user.
    n=int(input("Input a number to compute the factorial : "))
    print(factorial(n))
     # NOW print the function.
```

Output:

Input a number to compute the factorial : **6**

720



```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
# Taking input by user.
n=int(input("Input a number to compute the factiorial : "))
print(factorial(n))            # NOW print the function.
```

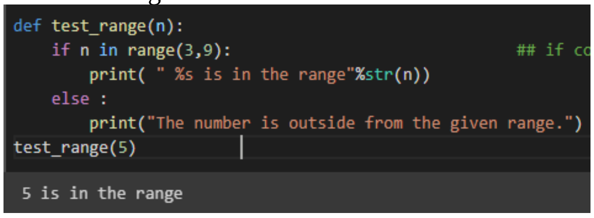
```
Input a number to compute the factiorial : 6
720
```

5. Write a Python function to check whether a number falls in a given range.

```
    # by defining a function.
    def  test_range(n):
    # Function defined "test_range" with one parameter "n".
        if n in range(3,9):
        # if condition applied to check whether number is in
range or not.
            print( " %s is in the range"%str(n))
        else :
            print("The number is outside from the given range.")
    test_range(5)
    # NOW print the function by giving value '5'.
```

Output:

5 is in the range



```
def test_range(n):
    if n in range(3,9):                              ## if co
        print( " %s is in the range"%str(n))
    else :
        print("The number is outside from the given range.")
test_range(5)
```

```
5 is in the range
```

6. Write a Python function that takes a list and returns a new list with unique elements of the first list.

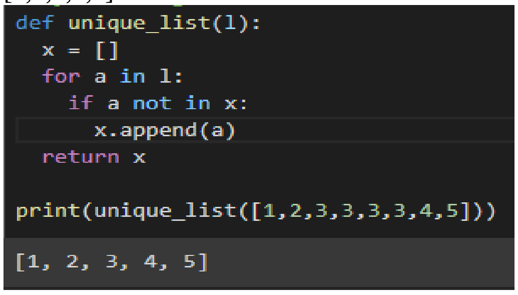
```
# by defining a function..
def  unique_list(l):
# Function defined “unique_list” with one parameter “l”.
  x = []
# an empty list create and store in “x”
  for a in l:
# for loop used to store “l” values in “a” respectively.
     if a not in x:
# checking, whether “a ” is in ‘List x’ or not.
        x.append(a)
```

 # Here **.append** function is used to merged list 's unique values in empty list 'x' using for loop.

```
  return x
print(unique_list([1,2,3,3,3,3,4,5]))
 # NOW print the function.
```

Output:

[1,2,3,4,5]



```
def unique_list(l):
  x = []
  for a in l:
    if a not in x:
      x.append(a)
  return x

print(unique_list([1,2,3,3,3,3,4,5]))
```

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

7. Write a Python function to find the Max of three numbers.

```
# by defining a function.
def numbers(n1,n2,n3):
```

# Function defined “numbers” with three parameters “n2,n2,n3”.

```
    if n1>n2 & n1>n3:
```

# if n1 is greater than n2 and n3, then n1 would be print as greater number.

```
      print("Max is ", n1)
    elif  n2>n1 & n2>n3:
```

# if n2 is greater than n1 and n3, then n2 would be print as greater number.

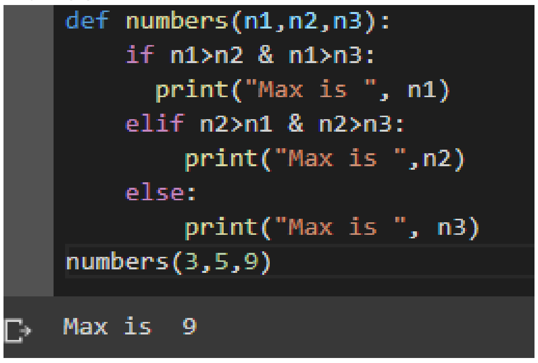
```
        print("Max is ",n2)
    else:
```

# above both condition ‘if & else’ are false then n3 would be print as greater number.

```
        print("Max is ", n3)
numbers(3,5,9)
 # calling Function  by giving values.
```

Output:

Max is 9



```
def numbers(n1,n2,n3):
    if n1>n2 & n1>n3:
      print("Max is ", n1)
    elif n2>n1 & n2>n3:
        print("Max is ",n2)
    else:
        print("Max is ", n3)
numbers(3,5,9)
```

```
Max is  9
```

8. Write a Python function to create and print a list where the values are square of numbers between 1 and 10 (both included).

```
# by defining a function.
def printValues():
   # Function defined “printValues”.
    lst = list()
# a List create and store in variable “lst”
    for i in range(1,11):
```

# for loop used to access 1 to 10 numbers with “Range” function.

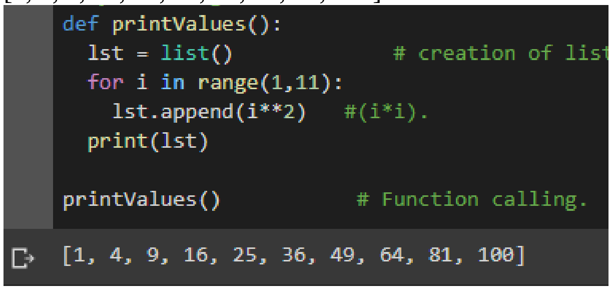
```
        lst.append(i**2)          #(i*i) can also be used.
```

# Here **.append** function used to merged lst with range' squares values using for loop.

```
      print(lst)
   printValues()
   # Function calling.
```

Output:

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]



```
def printValues():
  lst = list()             # creation of list
  for i in range(1,11):
    lst.append(i**2)    #(i*i).
  print(lst)

printValues()           # Function calling.
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

9. Write a Python function that takes a number as a parameter and check the number is prime or not.

# **Note** : A prime number (or a prime) is a natural number greater than 1 and that has no positive divisors other than 1 and itself.

```
   # by defining a function..
   def test_prime(n):
   # Function defined “test_prime” with one parameter “n”.
     if (n<=1):
```

# if condition applied to check whether “n” is less than or equal to 1 or not.

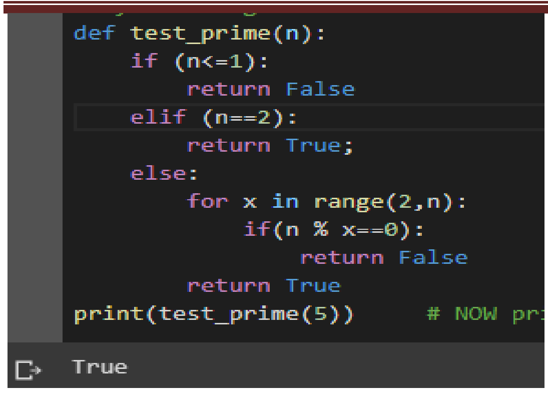
```
       return False
    # If  “n” is less than or equal to 1 then return false.
     elif (n==2):
       return True;
   # if “n” is equal to 2, return true.
     else:
       for x in range(2,n):
```

# for loop used to access 2 to n numbers with “Range” function.

```
         if(n % x==0):
         # if n is divisible by x then return false.
           return False
       return True
      # if n is not divisible by x then return true.
   print(test_prime(5))
   # Now print the function by giving of n value.
```

Output:

True



```
def test_prime(n):
    if (n<=1):
        return False
    elif (n==2):
        return True;
    else:
        for x in range(2,n):
            if(n % x==0):
                return False
        return True
print(test_prime(5))      # NOW pr
```

```
True
```

Lambda Function

rd Week lectures:

Submitted by: Zeeshan Hameed
Submitted to: Mr. Salman Mahmood Qazi & Muhammad Owais
Zeeshandatascience015@gmail.com

What is a Lambda Function in Python?

A lambda function is an anonymous function (i.e., defined without a name) that can take any number of arguments but, unlike normal functions, evaluates and returns only one expression. A lambda function in Python has the following syntax:
lambda parameters: expression
The anatomy of a lambda function includes three elements:

- The **keyword lambda** — an analog of def in normal functions
- The **parameters** — support passing positional and keyword arguments, just like normal functions
- The **body** — the expression for given parameters being evaluated with the lambda function

Note that, unlike a normal function, we don't surround the parameters of a lambda function with parentheses. If a lambda function takes two or more parameters, we list them with a comma.
We use a lambda function to evaluate only one short expression (ideally, a single-line) and only once, meaning that we aren't going to apply this function later. Usually, we pass a lambda function as an argument to a higher-order function (the one that takes in other functions as arguments), such as Python built-in functions like

- filter(),
- map(),
- reduce().

How a Lambda Function in Python Works

Let's look at a simple example of a lambda function:

```
lambda x: x + 1
```

Lambda Function Examples

```
# 1 Write a Python lamda function to find the Max of three
numbers
max=lambda x,y,z:print("maximum value is=",x) if (x>=y
and x>=z)\
   else(print("maximum value is=",y)if y>=x and y>=z
your range "))
enter2=int(input("enter second number
```

```
   else print("maximum value is=",z))
f=int(input("enter your first number"))
s=int(input("enter your second number"))
t=int(input("enter your third number"))
print(max(f,s,t))
#2 Write a Python lambda function to sum all the numbers
in a list
[1,3,4,50]
a=reduce(lafrom functools import reduce
li=mbda x,y:x+y,li)   # reduce() is to take an existing
function,
print(a)      #apply it cumulatively to all the items in an
iterable, and generate a single final value
#3 Write a Python lambda function to multiply all the
numbers in a list
from functools import reduce
li=[1,3,4,50]
b=reduce(lambda x,y:x*y,li)
print(b)
# 4 Write a Python program to reverse a
string.
reverse=lambda a:a[::-1]
enter=str(input("enter a string"))
print(reverse(enter))
#5 Write a Python lambda function to
calculate the factorial of a number (a
non-negative integer). The function
accepts the number as an argument
from functools import reduce
number=int(input('enter a number'))
if number<=0:
    print("enter a positive number")
else:
    c=reduce(lambda
x,y:x*y,range(1,number+1))
print(c)
# 6 Write a Python lambda function to
check whether a number falls in a given
range
enter1=int(input("enter first number of
```

```
your range "))
enter3=int(input("enter number to check
"))
rang=lambda :print("given number is
within range") if (enter3 in
range(enter1,enter2)) else print("number
is out of range")
print(rang())
# 7 Write a Python lambda function that
accepts a string and calculate the
number of upper case letters and lower
case letters
d=[]        # to store upper case
l=[]        # to store lower case
enter2=str(input("enter second number
your range "))
emp else None
for i in li:
   ulist(i)
print(emp)
# 9 Write a Python program that accepts
a hyphen-separated sequence of words as
input and prints the words in a hyphen-
separated sequence after sorting them
alphabetically.
li=lambda :[n for n in str(input('enter
a string with hyphen')).split("-")]
l=li()
l.sort()
print(l)
print('-'.join(l))
#10 Write a Python program to print the
even numbers from a given list using
lambda
w_list=[1,2,3,4,5,6]
li=list(filter(lambda l:l%2==0,w_list))
print(li)
#11 Write a Python program to access a
lambda inside a lambda

 pass
```

```
case=lambda i :d.append(i) if
i.isupper() else (l.append(i) if
i.islower() else None)
for i in enter2: # loop for iteration
string given in input
    case(i)
print('uprer',len(d))
print('uprer',len(l))
# 8 Write a Python lambda function that
takes a list and returns a new list with
unique elements of the first list.
emp=[]
li=[1,2,3,4,4,4,5,6,7]
ulist=lambda i:emp.append(i) if i not in
l=lambda x:lambda y:x+y
first=l(5)
print(first(10))
#12 Write a Python function to check
whether a number is perfect or not.
from functools import reduce
# for enter in range(2,10000):  # this
loop can be used to find perfact number
in given range
enter = int(input("enter a number"))
flist = list(filter(lambda x: enter % x
== 0, range(1, enter)))
# print(l)
fac_sum = reduce(lambda x, y: x + y,
flist)
if fac_sum == enter:
    print(enter, ' is a perfect number
')
else:
    print(enter, 'is not a perfect
number')
```

Introduction to object-oriented programming

Ali Raza
6th Week
Submitted to Mr Salman Qazi & Muhammad Owais Saleem
Aliraza328333@gmail.com

Introduction

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects" that can contain data and code. The data is often implemented as attributes. Functions implement the associated code for the data and are usually referred to in object-oriented jargon as methods. In OOP, computer programs are designed by being made up of objects that interact with each other via the methods.

Object-Oriented means directed toward objects. In other words, it means functionally directed towards modeling objects. This is one of the many techniques used for modeling complex systems by describing a collection of interacting objects via their data and behavior. Python, an Object-Oriented Programming (OOP), is a way of programming that focuses on using objects and classes to design and build applications. Major pillars of Object-Oriented Programming (OOP) are Inheritance, Polymorphism, Abstraction, ad Encapsulation. Object-Oriented Analysis(OOA) is the process of examining a problem, system, or task and identifying the objects and interactions between them.

Why Choose Object-oriented Programming?

Python was designed with an object-oriented approach. OOP offers the following advantages:

- Provides a clear program structure, which makes it easy to map real-world problems and their solutions.
- Facilitates easy maintenance and modification of existing code.
- Enhances program modularity because each object exists independently, and new features can be added easily without disturbing the existing ones.
- Presents a good framework for code libraries where supplied components can be easily adapted and modified by the programmer.
- Imparts code reusability

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by their class) for modifying their state.

Let us understand each of the pillars of object-oriented programming in brief:

Encapsulation

This property hides unnecessary details and makes it easier to manage the program structure. Each object's implementation and state are hidden behind well-defined boundaries and that provides a clean and simple interface for working with them. One way to accomplish this is by making the data private.

Inheritance

Inheritance, also called generalization, allows us to capture a hierarchal relationship between classes and objects. For instance, a 'fruit' is a generalization of 'orange'. Inheritance is very useful from a code reuse perspective.

Abstraction

This property allows us to hide the details and expose only the essential features of a concept or object. For example, a person driving a scooter knows that by pressing a horn, the sound is emitted, but he has no idea about how the sound is generated by pressing the horn.

Polymorphism

Polymorphism means many forms. That is, a thing or action is present in different forms or ways. One good example of polymorphism is constructor overloading in classes

Object-Oriented Python

The heart of Python programming is object and OOP, however, you need not restrict
yourself to use the OOP by organizing your code into classes. OOP adds to the whole
design philosophy of Python and encourages a clean and pragmatic way to programming.
OOP also enables in writing of bigger and more complex programs.

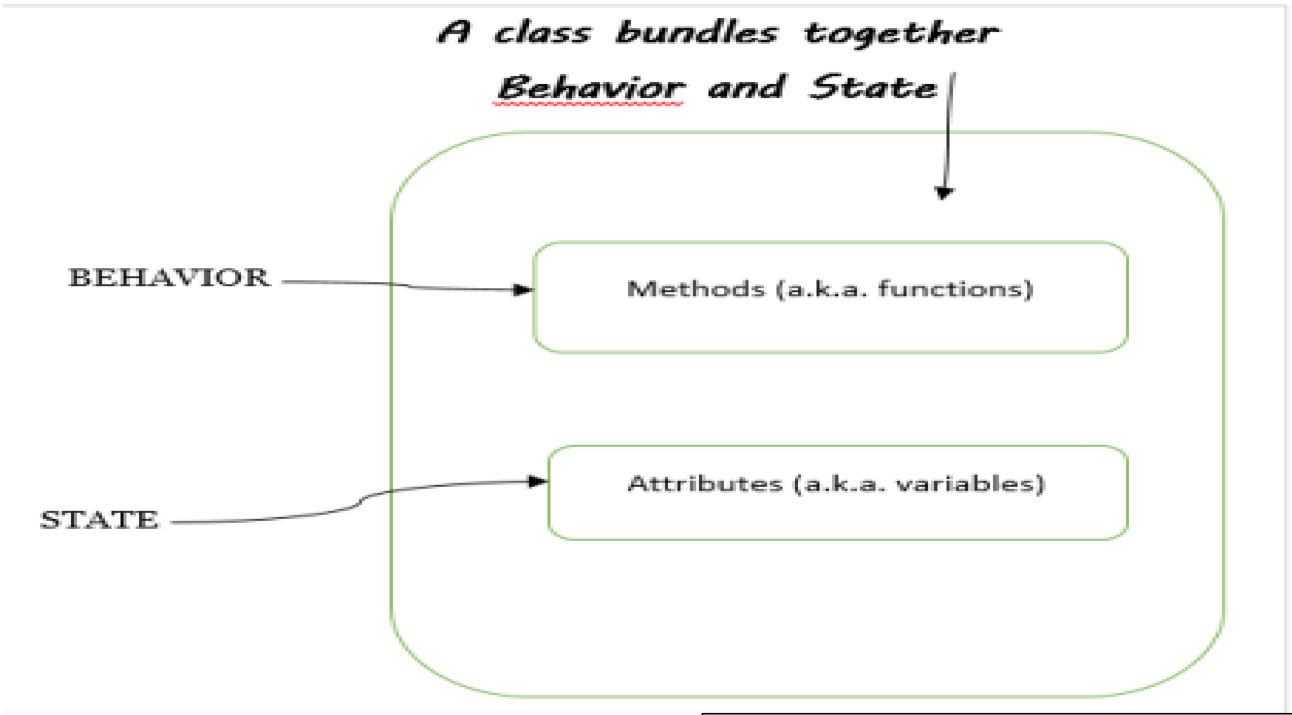
Class

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

```
class ClassName:
   # Statement-1
   .
   .
   .
   # Statement-N
```

Class Bundles: Behavior and State

A class will let you bundle together the behavior and state of an object. Observe the following diagram for better understanding:



The following points are worth notable when discussing class bundles:

• The word **behavior** is identical to **function** – it is a piece of code that does something (or implements a behavior)

• The word **state is** identical to **variables** – it is a place to store values within a class.

• When we assert a class behavior and state together, it means that a class packages functions and variables.

Classes have methods and attributes

In Python, creating a method defines a class behavior. The word method is the OOP name
given to a function that is defined within a class. To sum up:

Class functions is a synonym for **methods**
Class variables are a synonym for name **attributes**.
Class – a blueprint for an instance with exact behavior.
Object – one of the instances of the class, perform functionality defined in the class.
Type – indicates the class the instance belongs to
Attribute – Any object value: object.attribute
Method – a "callable attribute" defined in the class

Observe the following piece of code for example:

```
var = “Hello, John”
print( type (var))        # < type ‘str’> or <class 'str'>
print(var.upper())        # upper() method is called,
HELLO, JOHN
```

Creation and Instantiation

The following code shows how to create our first class and then its instance.

```
class MyClass():
 pass
# Create first instance of MyClass
this_obj = MyClass()
print(this_obj)
# Another instance of MyClass
that_obj = MyClass()
print (that_obj)
```

Here we have created a class called **MyClass** and which does not do any tasks**. pass** in the above code indicates that this block is empty, that is it is an empty class definition.

Let us create an instance **this_obj** of **MyClass()** class and print it as shown:

Output

```
<__main__.MyClass object at 0x03B08E10>
<__main__.MyClass object at 0x0369D390>
```

Here, we have created an instance of **MyClass.** The hex code refers to the address where the object is being stored. Another instance is pointing to another address.

Now let us define one variable inside the class **MyClass()** and get the variable from the instance of that class as shown in the following code:

```
class MyClass(object):
 var = 9
# Create first instance of MyClass
this_obj = MyClass()
print(this_obj.var)
# Another instance of MyClass
that_obj = MyClass()
print (that_obj.var)
```

Output

You can observe the following output when you execute the code given above:

```
9
9
```

self

Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it

Instance Methods

A function defined in a class is called a **method**. An instance method requires an instance in order to call it and requires no decorator. When creating an instance method, the first parameter is always **self**. Though we can call it (self) by any other name, it is recommended to use self, as it is a naming convention.

```
class MyClass():
 var=9
 def firstM(self):
 print("hello, World")
obj = MyClass()
print(obj.var)
obj.firstM()
```

Output

You can observe the following output when you execute the code given above:

```
9
hello, World
```

Note that in the above program, we defined a method with self as argument. But we cannot call the method as we have not declared any argument to it

```
class MyClass(object):
 def firstM(self):
 print("hello, World")
 print(self)
obj = MyClass()
obj.firstM()
print(obj)
```

Output

You can observe the following output when you execute the code given above:

```
hello, World
<__main__.MyClass object at 0x036A8E10>
<__main__.MyClass object at 0x036A8E10>
```

__Init__ Constructor

The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created. In Python the __init__() method is called the constructor and is always called when an object is created.

Syntax of constructor declaration :

```
def __init__(self):

    # body of the constructor
```

Class with Constructor:

```
class Myclass:
    def __init__(self):
        print("haji zeshan")
x=Myclass()     # object of a class
```

The __init__() function is called automatically every time the class is being used to create a new object.

Output

```
haji zeshan
```

Types of constructors :

default constructor: The default constructor is a simple constructor which doesn't accept any arguments.

```
class Myclass:
    def __init__(self):
        print("haji zeshan")
x=Myclass()     # object of a class
```

parameterized constructor: constructor with parameters is known as parameterized constructor.

```
class myclass():
 def __init__(self,aaa, bbb):
 self.a = aaa
 self.b = bbb
x = myclass(4.5, 3)
print(x.a, x.b)
```

Output

```
4.5 3
```

Working with Class and Instance Data

We can store data either in a class or in an instance. When we design a class, we decide which data belongs to the instance and which data should be stored into the overall class.

An instance can access the class data. If we create multiple instances, then these instances can access their individual attribute values as well the overall class data.

Thus, a class data is the data that is shared among all the instances. Observe the code given below for better undersanding:

```
class cal():
    def __init__(self,a,b):          # init function is
automatically called by calling class
        self.a=a
        self.b=b
    def add(self):              # sum function
        return self.a+self.b
    def mul(self):               # multiplication function
        return self.a*self.b
    def div(self):               # division  function
        return self.a/self.b
    def sub(self):                   # subtraction function
        return self.a-self.b
a=int(input("Enter first number: "))      # take values
from user
b=int(input("Enter second number: "))
# take values from user
obj=cal(a,b)       # calling class
choice=1
while choice!=0:       #loop for multiple  calculation
    print("0. Exit")
    print("1. Add")
    print("2. Subtraction")
    print("3. Multiplication")
    print("4. Division")
    choice=int(input("Enter choice: "))
    if choice==1:
        print("Result: ",obj.add())
    elif choice==2:
        print("Result: ",obj.sub())
    elif choice==3:
        print("Result: ",obj.mul())
    elif choice==4:
        print("Result: ",round(obj.div(),2))
    elif choice==0:
        print("Exiting!")
    else:
        print("Invalid choice!!")
```

Output

```
Enter first number: 5
Enter second number: 10
0. Exit
1. Add
2. Subtraction
3. Multiplication
4. Division
Enter choice: 1
Result:  15
0. Exit
1. Add
2. Subtraction
3. Multiplication
4. Division
```

Inheritance in Python

Inheritance is the capability of one class to derive or inherit the properties from another class. The benefits of inheritance are:

- It represents real-world relationships well.
- It provides reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Parent class is the class being inherited from, also called base class.
Child class is the class that inherits from another class, also called derived class.
Create a Parent Class
Any class can be a parent class, so the syntax is the same as creating any other class:
Example

Create a class named Person,
with firstname and lastname properties, and
a printname method:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe")
x.printname()
```

Output
Johan Doe

Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:
Example
Create a class named Student, which will inherit the properties and methods from the Person class:

```
class Student(Person):
  pass
```

Now the Student class has the same properties and methods as the Person class.

Python pass Statement

When the pass statement is executed, nothing happens, but you avoid getting an error when empty code is not allowed. Empty code is not allowed in loops, function definitions, class definitions, or in if statements.

Example

Use the Student class to create an object, and then execute the print name method:

```
x = Student("Mike", "Olsen")
x.printname()
```

```
# Python code to demonstrate how parent constructors
# are called.

# parent class
class Person( ):

    # __init__ is known as the constructor
    def __init__(self, name, idnumber):
        self.name = name
        self.idnumber = idnumber
    def display(self):
        print(self.name)
        print(self.idnumber)

# child class
class Employee( Person ):
        def __init__(self, name, idnumber, salary, post):
            self.salary = salary
            self.post = post

            # invoking the __init__ of the parent class
            Person.__init__(self, name, idnumber)


# creation of an object variable or an instance
a = Employee('Rahul', 886012, 200000, "Intern")

# calling a function of the class Person using its
instance
a.display()
```

Output

```
Rahul
886012
```

Add the __init__() Function

So far 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 (instead of the pass keyword).

The __init__() function is called automatically every time the class is being used to create a new object.

Example

Add the `__init__()` function to the `Student` class:

```
class Student(Person):
  def __init__(self, fname, lname):
    #add properties etc.
```

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 Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

Now we have successfully added the __init__() function, and kept the inheritance of the parent class, and we are ready to add functionality in the __init__() function.

Different forms of Inheritance:

1. Single inheritance: When a child class inherits from only one parent class, it is called single inheritance. We saw an example above.

2. Multiple inheritance: When a child class inherits from multiple parent classes, it is called multiple inheritance.

```
# Python example to show the working of multiple
# inheritance
class Base1():
  def __init__(self):
    self.str1 = "Geek1"
    print("Base1")

class Base2(object):
  def __init__(self):
    self.str2 = "Geek2"
    print("Base2")

class Derived(Base1, Base2):
  def __init__(self):

    # Calling constructors of Base1
    # and Base2 classes
    Base1.__init__(self)
    Base2.__init__(self)
    print("Derived")

  def printStrs(self):
    print(self.str1, self.str2)


ob = Derived()
ob.printStrs()
```

Output

```
Base1
Base2
Derived
Geek1 Geek2
```

Use the super() Function

Python also has a super() function that will make the child class inherit all the methods and properties from its parent:

Example

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

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.

Add Properties

Example

Add a property called graduationyear to the Student class:

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
```

In the example below, the year 2019 should be a variable, and passed into the Student class when creating student objects. To do so, add another parameter in the __init__() function:

Example

Add a year parameter, and pass the correct year when creating objects:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
```

Add Methods

Example

Add a method called welcome to the Student class:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname,
self.lastname, "to the class of",
self.graduationyear)
```

Example

```
# Python program to demonstrate
# super function


class Animals:

    # Initializing constructor
    def __init__(self):
        self.legs = 4
        self.domestic = True
        self.tail = True
        self.mammals = True

    def isMammal(self):
        if self.mammals:
            print("It is a mammal.")

    def isDomestic(self):
        if self.domestic:
            print("It is a domestic animal.")

class Dogs(Animals):
    def __init__(self):
        super().__init__()

    def isMammal(self):
        super().isMammal()

class Horses(Animals):
    def __init__(self):
        super().__init__()

    def hasTailandLegs(self):
        if self.tail and self.legs == 4:
            print("Has legs and tail")
```

```
# Driver code
Tom = Dogs()
Tom.isMammal()
Bruno = Horses()
Bruno.hasTailandLegs()
```

Output

```
It is a mammal.

Has legs and tail
```

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

Encapsulation in Python

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as **private variables.**

A **class** is an example of **encapsulation** as it encapsulates all the data that is member functions, variables, etc.

define a private member prefix the member name with double underscore "__".

```
# Python program to
# demonstrate protected members

# Creating a base class
class Base:
    def __init__(self):

        # Protected member
        self._a = 2

# Creating a derived class
class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling protected member of base class: ",
              self._a)

        # Modify the protected variable:
        self._a = 3
        print("Calling modified protected member outside class: ",
              self._a)


obj1 = Derived()
```

```
obj2 = Base()

# Calling protected member
# Can be accessed but should not be done due to convention
print("Accessing protected member of obj1: ", obj1._a)

# Accessing the protected variable outside
print("Accessing protected member of obj2: ", obj2._a)
```

Output:

```
Calling protected member of base class:  2
Calling modified protected member outside class:  3
Accessing protected member of obj1:  3
Accessing protected member of obj2:
```

Polymorphism in Python

The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types.

Example of inbuilt polymorphic functions :

```
# Python program to demonstrate in-built poly-
# morphic functions

# len() being used for a string
print(len("geeks"))

# len() being used for a list
print(len([10, 20, 30]))
```

Output:

```
5
3
```

Examples of user-defined polymorphic functions :

```
# A simple Python function to demonstrate
# Polymorphism

def add(x, y, z = 0):
    return x + y+z

# Driver code
print(add(2, 3))
print(add(2, 3, 4))
```

Output:

```
5
9
```

Polymorphism with class methods:

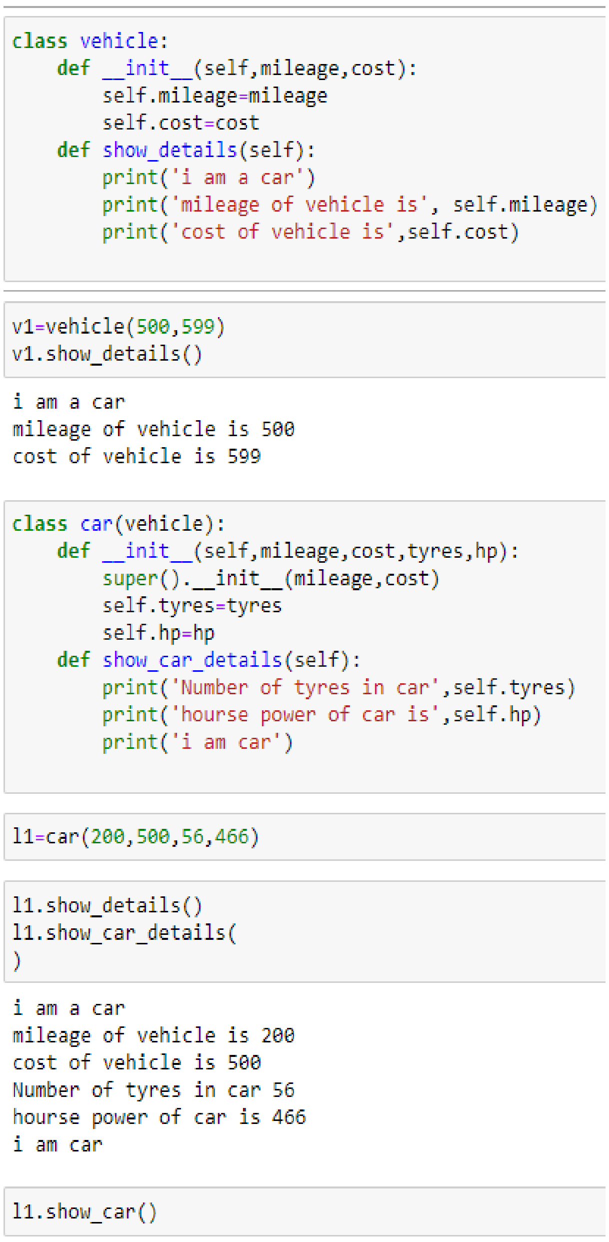
The below code shows how Python can use two different class types, in the same way. We create a for loop that iterates through a tuple of objects. Then call the methods without being concerned about which class type each object is. We assume that these methods actually exist in each class.

```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of India.")

    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")

obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()
    country.type()
```

Output:

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

Example of inheritance



```
class vehicle:
    def __init__(self,mileage,cost):
        self.mileage=mileage
        self.cost=cost
    def show_details(self):
        print('i am a car')
        print('mileage of vehicle is', self.mileage)
        print('cost of vehicle is',self.cost)
```

```
v1=vehicle(500,599)
v1.show_details()
```

```
i am a car
mileage of vehicle is 500
cost of vehicle is 599
```

```
class car(vehicle):
    def __init__(self,mileage,cost,tyres,hp):
        super().__init__(mileage,cost)
        self.tyres=tyres
        self.hp=hp
    def show_car_details(self):
        print('Number of tyres in car',self.tyres)
        print('hourse power of car is',self.hp)
        print('i am car')
```

```
l1=car(200,500,56,466)
```

```
l1.show_details()
l1.show_car_details(
)
```

```
i am a car
mileage of vehicle is 200
cost of vehicle is 500
Number of tyres in car 56
hourse power of car is 466
i am car
```

```
l1.show_car()
```