Python and Advance Python-

What is Python?

-Python is a 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,
- System scripting.

What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

Good to know

- The most recent major version of Python is Python 3. However, Python 2, although not being updated with anything other than security updates, is still quite popular.
- Python will be written in a text editor. It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

Python Syntax compared to other programming languages

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

#Advantage of python??

1. Easy to Read, Learn and Write

1. Easy to Read, Learn and Write

-Python is a high-level programming language that has English-like syntax. This makes it easier to read and understand the code.
-Python is really easy to pick up and learn, that is why a lot of people recommend Python to beginners. You need less lines of code to perform the same task as compared to other major languages like C/C++ and Java.

2. Improved Productivity

-Python is a very productive language. Due to the simplicity of Python, developers can focus on solving the problem. They don't need to spend too much time in understanding the syntax or behavior of the programming language. You write less code and get more things done.

3. Interpreted Language

-Python is an interpreted language which means that Python directly executes the code line by line. In case of any error, it stops further execution and reports back the error which has occurred.
-Python shows only one error even if the program has multiple errors. This makes debugging/problem solving easier.

4. Dynamically Typed

-Python doesn't know the type of variable until we run the code. It automatically assigns the data type during execution. The programmer doesn't need to worry about declaring variables and their data types.

5. Free and Open-Source

-Python comes under the OSI approved open-source license(OSI approval means you **have the unconditional right to use the software in question for any purpose** (sometimes calls "freedom zero"). This makes it free to use and distribute. You can download the source code, modify it and even distribute your version of Python. This is useful for organizations that want to modify some specific behavior and use their version for development.

6. Vast Libraries Support

-The standard library of Python is huge, you can find almost all the functions needed for your task. So, you don't have to depend on external libraries.
-But even if you do, a Python package manager (pip) makes things easier to import other great packages from the Python package index (PyPi). It consists of over 200,000 packages.

7. Portability

-In many languages like C/C++, you need to change your code to run the program on different platforms. That is not the same with Python. You only write once and run it anywhere.
However, you should be careful not to include any system-dependent features.

#Disadvantages...??

1. Slow Speed

-We discussed above that Python is an interpreted language and dynamically-typed language. The line by line execution of code often leads to slow execution.
-The dynamic nature of Python is also responsible for the slow speed of Python because it has to do the extra work while executing code. So, Python is not used for purposes where speed is an important aspect of the project.

2. Not Memory Efficient

-To provide simplicity to the developer, Python has to do a little tradeoff. The Python programming language uses a large amount of memory. This can be a disadvantage while building applications when we prefer memory optimization.

3. Weak in Mobile Computing

-Python is generally used in server-side programming. We don't get to see Python on the client-side or mobile applications because of the following reasons. Python is not memory efficient and it has slow processing power as compared to other languages.

4. Database Access

-Programming in Python is easy and stress-free. But when we are interacting with the database, it lacks behind.
-The Python's database access layer is primitive and underdeveloped in comparison to the popular technologies like JDBC and ODBC.
-Huge enterprises need smooth interaction of complex legacy data and Python is thus rarely used in enterprises.

5. Runtime Errors

-As we know Python is a dynamically typed language so the data type of a variable can change anytime. A variable containing integer number may hold a string in the future, which can lead to Runtime Errors.
-Therefore Python programmers need to perform thorough testing of the applications.

Summary

-Python is a simple, versatile and a complete programming language. It is a great choice for beginners up to professionals. Although it has some disadvantages, we can observe that the advantages exceed the disadvantages. Even Google has made Python one of its primary programming languages.

#High level & Low level language-

-The main difference between **high level language** and **low level language** is that, Programmers can easily understand or interpret or compile the high level language in comparison of machine. On the other hand, Machine can easily understand the low level language in comparison of human

beings.

High Level Language	Low Level Language
These are Interpreted	Direct memory management
They have open classes and message-style methods which are known as Dynamic constructs	Hardware has extremely little abstraction which is actually close to having none.
Poor performance	Much fast than high level
Codes are Concise	Statements correspond directly to clock cycles
Flexible syntax and easy to read	Superb performance but hard to write
Is object oriented and functional	Few support and hard to learn
Large community	

-High level-Human can operate
-Low level-Machine operated
-With the help of compiler we can convert High level to low level & also check syntax.
-in python compiler not available.
-C,C++ and java languages called as compiler based language.

#What is mean by POC
-Proof of concept
-A proof of concept (POC) is a demonstration to verify that certain concepts or theories have the potential for real-world application. In a nutshell, a POC represents the evidence demonstrating that a project or product is feasible and worthy enough to justify the expenses needed to support and develop it.
-POC is therefore a prototype that is designed to determine feasibility, but does not represent deliverables. It is usually required by investors who need tangible proof that a startup and its business proposal can guarantee a healthy return on investment (ROI).

Note-Project are measured in hours

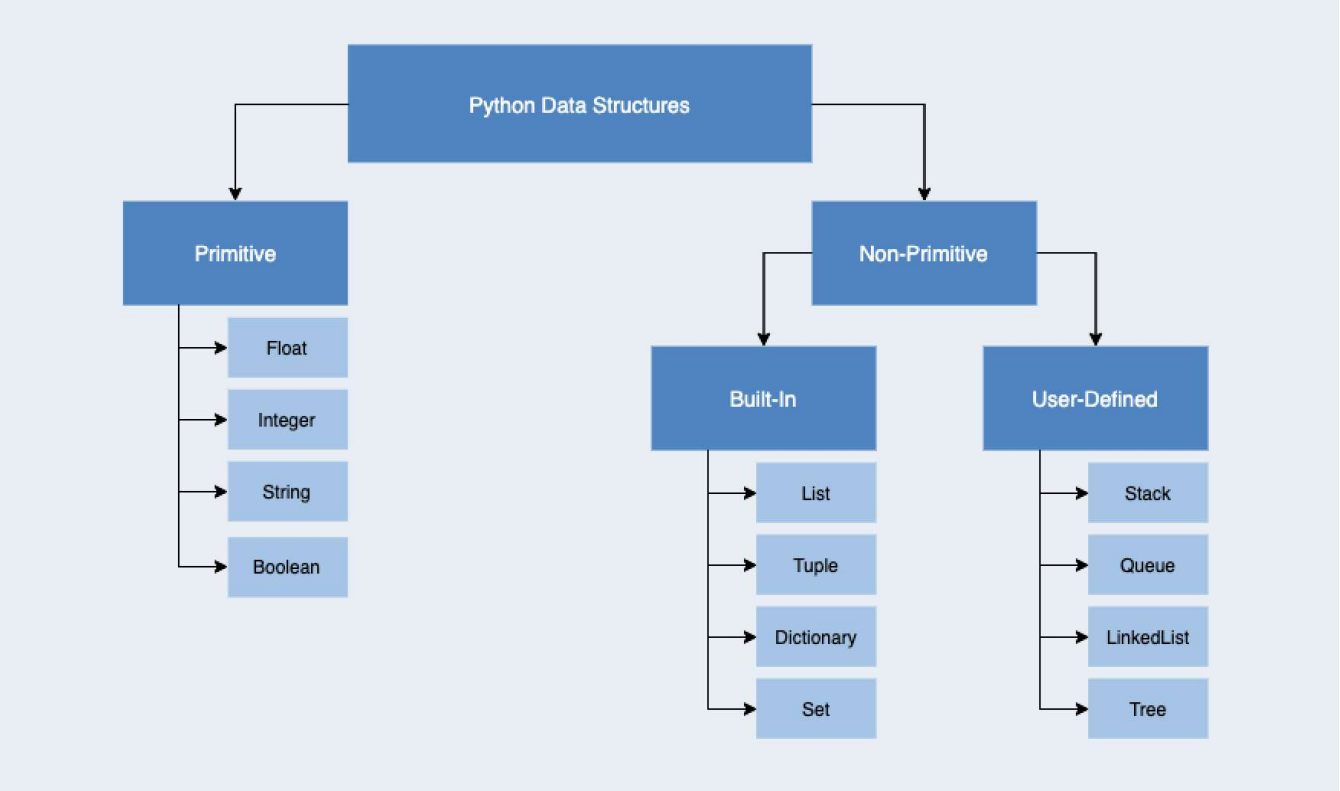
#Data type & Data structure-

1)Data type-
-Complete number
-Fraction number
-Character/String(character data type is not present in python)

2)Data structure-
-Arranging and storing data in particular manner called as "Data Structure"
-Python is an object-oriented programming (OOP) language. Classes and objects are used to structure and modularize code to be reusable and easy to modify. OOP requires the use of **data structures** to organize and store data in a way that can be efficiently accessed.
-Python has **primitive** (or basic) data structures such as floats, integers, strings, and Booleans.

Python also has **non-primitive** data structures such as lists, tuples, dictionaries, and sets.
-Non-primitive data structures store a collection of values in various formats rather than a single value. Some can hold data structures within the data structure, creating depth and complexity in data storage capabilities.

-Types of data structure-



 # Python IDE

Dev	DA/ML/DL
Pycharm(Mostly used)	Jupyter Notebook
vscode	
spyder	
notepad++	

#Mutable & Immutable Data structure-

-If the value can change, the object is called **mutable**, while if the value cannot change, the object is called **immutable**.

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓

list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

i.e Set,Dictionary and List is mutable.

#Python Versions-

Version	Release Date	Important Features
Python 1.0	January 1994	• Functional programming tools (lambda, map, filter and reduce). • Support for complex numbers.
Python 2.0	October 2000	• List comprehension. • Cycle-detecting garbage collector. • Support for Unicode. Unification of data types and classes
Python 2.7.0	July 2010	
Python 3	December 2008	• Backward incompatible. • print keyword changed to print() function • raw_input() function depreciated • Unified str/Unicode types. • Utilities for automatic conversion of Pytthon 2.x code
Python 3.6	December 2016	
Python 3.6.5	March 2018	
Python 3.7.0	May 2018	• New C API for thread-local storage • Built-in breakpoint() • Data classes
Python 3.8	October 2019	• Assignment Expression • Positional-only parameters • Parallel filesystem cache for compiled bytecode files
Python 3.9 - Current Version	October 2020	• Dictionary Merge & Update Operators • New removeprefix() and removesuffix() string methods • Builtin Generic Types

---

#Data Structure-

1)List=[ ]

a.It is collection of similar or different type of data types.

b.its symbol is [ ]

b.its symbol is [ ]
c.Homogeneous & Heterogeneous
d.It is mutable type ,eg. We can perform insert/delete/update operation.
e.Lists can have duplicate elements.
f.it is ordered data type.(so indexing is available/or we can perform indexing).
#How to list store in memory..??
-Every elements are stored in indexing.
-Indexing start with "Zeroth"
-Indexing is use to access and perform different operations.

#Indexing Types-
1) Forward Indexing-
-FI is always start from left to right.
-First element is 0(Zeroth Index)

2) Reverse Indexing-
-RI is always start from right to Left.
-5 -4 -3 -2 - 1

#Code Writing Techniques-
#Types of cases
1.Pascal Case
First letter and middle letter CAPITAL
e.g PhonePay, LinkedIn

2.Camel Case
-typographical convention in which an initial capital is used for the first letter of a word forming the second element of a closed compound, e.g. *PayPal*, *iPhone*, *MasterCard*,eBay

3.snake_case
-Using underscore between 2 letters.
e.g "foo_bar", "hello_world"

Note-While writing a list variables-
P = [ ](Square bracket is compulsory)

#Python List Methods
append() and extend()

-The **append()** method allows you to add another item to the end of your list. The method takes *one required argument*, which is the item you wish to add to your list.

```
Syntax: list.append(item)
toppings = ['pepperoni', 'sausage', 'mushroom']
toppings.append('onion')
```

```
--> ['pepperoni', 'sausage', 'mushroom', 'onion']
```

-The **extend()** method is similar to append() in that it allows you to add onto your list; however, the extend() method allows you to add all of the items from another iterable (list, tuple, set, etc.) to the end of your list as separate items instead of one item. The method takes *one required argument*, the iterable.

```
Syntax: list.extend(iterable)
toppings = ['pepperoni', 'sausage', 'mushroom']
more_toppings = ['onion', 'bacon']
toppings.extend(more_toppings)
--> ['pepperoni', 'sausage', 'mushroom', 'onion', 'bacon']

To contrast...
toppings.append(more_toppings)
--> ['pepperoni', 'sausage', 'mushroom', ['onion', 'bacon']]
```

As you can see above, when the extend() method is used with an iterable, *each item* in the iterable is added to the list as *separate items* no longer bounded. On the contrary, when the append() method is used with an iterable as the argument, the *entire* iterable is added to the list as *one item*. It's always important to pay close attention to the commas and brackets present in a list.

pop() and remove()

The **pop()** method allows you to remove an element from your list at a specified index value. The method can take *one optional argument*, the integer value of the index you wish to remove — by default, pop() will remove the last item in the list, as the default value is -1.

Note-

1. ***Pop method is used where we want to use our list like a stack or a queue.***
2. ***With the help of pop method we can delete last end item/value and also can fetch deleted item/value from list.***

```
Syntax: list.pop(index)
toppings = ['pepperoni', 'sausage', 'mushroom']
toppings.pop(1)
--> ['pepperoni', 'mushroom']
toppings.pop()
--> ['pepperoni', 'sausage']

If you wanted to retrieve the removed item...
extra = toppings.pop(1)
extra --> 'sausage'
```

Like the pop() method, the **remove()** method allows you to remove an item from your list. The remove() method, though, removes the first occurrence of a specified value in a list. The method takes *one required argument*, the item you wish to remove.

```
Syntax: list.remove(item)
```

```
toppings = ['pepperoni', 'sausage', 'mushroom']
toppings.remove('sausage')
--> ['pepperoni', 'mushroom']
```

For both pop() and remove(), if the argument is out of range or does not exist in the list, respectively, you will get an error.

sort() and reverse()

The **sort()** method sorts a list by certain criteria. The method can take *two optional arguments*. The first argument is setting either *reverse=True* or *reverse=False*. By default, this argument is set to *reverse=False*, which will result in alphabetical order if the list consists of only strings, or ascending order if the list consists of only numbers. The second argument allows you to set a *key=* to a function that you can use to specify how exactly you want your list sorted if it's more complex than the default ordering that the sort() method does. This could be a built-in Python function, a function you defined elsewhere in your program, or an in-line lambda function that you write.

Note-Sort will arrange string values at alphabetical order and number as ascending order by default.

```
Syntax: list.sort(reverse=True|False, key=function)
toppings = ['pepperoni', 'sausage', 'mushroom']
toppings.sort()
--> ['mushroom', 'pepperoni', 'sausage']
toppings.sort(reverse=True)
--> ['sausage', 'pepperoni', 'mushroom']
toppings.sort(reverse=True, key=lambda x: len(x))
--> ['pepperoni', 'mushroom', 'sausage']
* Sorted in reverse order by length of the topping name

prices = [1.50, 2.00, 0.50]
prices.sort(reverse=False)
--> [0.50, 1.50, 2.00]
prices.sort(reverse=True)
--> [2.00, 1.50, 0.50]


pies = [['bacon', 'ranch'], ['sausage', 'peppers']]
pies.sort(reverse=True)
--> [['sausage', 'peppers'], ['bacon', 'ranch']]
* Sorts iterators by their first value
```

The **reverse()** method simply reverses the order of the items in the list. The method takes *no arguments*.

```
Syntax: list.reverse()
toppings = ['pepperoni', 'sausage', 'mushroom']
toppings.reverse()
--> ['mushroom', 'sausage', 'pepperoni']

prices = [1.50, 2.00, 0.50]
prices.reverse()
--> [0.50, 2.00, 1.50]
```

count()

The **count()** method returns the number of occurrences of a specified item in a list. The method takes *one required argument*, which is the item you wish to find the count of. This method can be useful if you wish to find out what items appear more than once in a list.

```
Syntax: list.count(item)
toppings = ['pepperoni', 'sausage', 'mushroom', 'sausage']
toppings.count('sausage')
--> 2
toppings.count('pepperoni')
--> 1
toppings.count('bacon')
--> 0
```

index()

The **index()** method returns the index of the first occurrence of the specified item. The method takes *one required argument*, which is the item whose index you wish to find. If the item does not exist in the list, you will get an error.

```
Syntax: list.index(item)
toppings = ['pepperoni', 'sausage', 'mushroom', 'sausage']
toppings.index('mushroom')
--> 2
toppings.index('pepperoni')
--> 0
```

insert()

The **insert()** method inserts a specified item into a list at a specified index. The method takes *two required arguments* — the integer index you wish to insert the value at and the item you'd like to insert.

```
Syntax: list.insert(index, index)
toppings = ['pepperoni', 'sausage', 'mushroom']
toppings.insert(1, 'onion')
--> ['pepperoni', 'onion', 'sausage', 'mushroom']
```

copy()

The **copy()** method simply returns a copy of your list. The method takes *no arguments*.

```
Syntax: list.copy()
toppings = ['pepperoni', 'sausage', 'mushroom']
toppings2 = toppings.copy()
toppings2 --> ['pepperoni', 'sausage', 'mushroom']
```

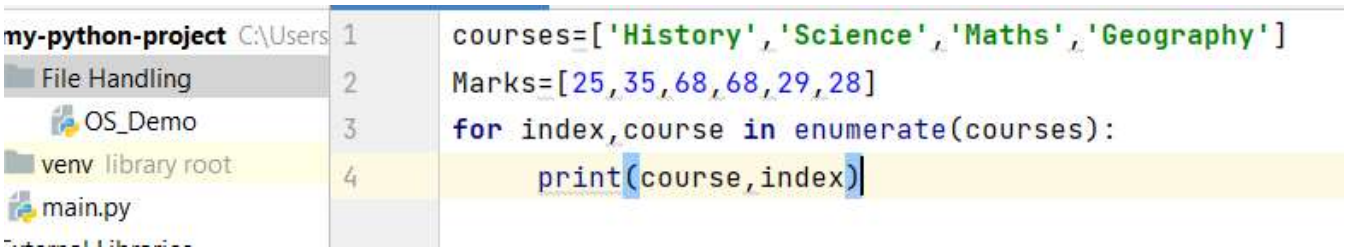
clear()

The **clear()** method simply removes all items from a list, leaving an empty list. The method takes *no arguments*.

```
Syntax: list.clear()
toppings = ['pepperoni', 'sausage', 'mushroom']
toppings.clear()
--> []
```

Note-Interview question

- ***How we can fetch list values with index number...??***



-Summary Of commands

-Insert-To add element at desired location
-Append-No of elements
-Extend-To add elements at one end

#Remove list items
-Pop- Delete using index
-Remove-Delete using element
-Copy-create same copy
-Count-return no of occurrence
-Index-return index number of element
-sort -sort(reverse=True)desc
-reverse-Simply reverse the list.

---

2)Tuple-

1. -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.
2. It is immutable-Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.
3. Read only type.
4. Used where high security required(we don't update most frequently).
5. It is ordered data type.(so indexing is available/or we can perform indexing).When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.
6. 0 index.
7. Denoted as ( ).
8. Duplicates are allowed-Since tuples are indexed, they can have items with the same value.
9. We can only perform two types of methods i.e- count & index.

3) Set-

1. Set are used to store multiple items in a single variable.
2. Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List , Tuple, and Dictionary, all with different qualities and usage.
3. If you trying to make empty set ,it will show dictionary type.
4. set ={at least one element}
5. Collection of unique items.
6. Indexing is not possible.
7. Duplicates are not allowed.
8. Set in unordered(Unordered means that the items in a set do not have a defined order.)
9. It is mutable type.

Methods in set-

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

4)Dictionary-

1. Collection of key -value pair(Called as Item).
2. If key not found ,Get method will provide default message.
3. Key should be immutable.value can be mutable or immutable.
4. with the help of list we can create dictionary

4. With the help of list we can create dictionary.
5. Interview question-Can we assign List/Set/Dict as key??-we can't assign list as a key.i.e key is immutable and Dict/List and Set are mutable.
6. With the help of list we can add key and values i.e item in dictionary.
7. Can we assign Tuple as a Key?? - Yes Tuple is mutable so we can assign as a Key.but it is not recommended.
8. 0Changeable-Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.
9. We can assign string,numbers and tuples as a Key in dictionary .
10. With the help of 'update'we can add 2 or more items at a time.
11. Pop-by using key you delete value with the help of key.
12. As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.

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

Python Collections (Arrays)

-There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered* and changeable. No duplicate members.

Interview questions-

- What is Shallow copy and deep copy...???
- What is nested list or nested object..??

#Type Casting-

1. Type Casting is the method to convert the variable data type into a certain data type in order to the operation required to be performed by users.
2. Also we can say, converting of values one data type into another data type.

3. Conversions of data types or data structure.
   i.e

   String to Integer
   Integer to String
   String to Float
   List to Tuple
   Tuple to List
   Dictionary to List(zip function required)

#String -

1. Like many other popular programming languages, strings in Python are arrays of bytes representing uni code characters.However, Python does not have a character data type, a single character is simply a string with a length of 1
2. It is Immutable data structure.
3. It is ordered type ,we can perform indexing and as well as slicing..
4. It is zero based data structure.
5. String is case sensitive.if you give define input as a string and you give condition then input and and condition should have same case.
6. *In find method-If specified value is not available in string it will return with "-1".find returns index number of specified value.*
7. Square brackets can be used to access elements of the string.
8. What is difference between index and find method in string..?? - both have same functions only difference is in output i,e FIND-If specified value is not available in string it will return with "-1" & in INDEX-If specified value is not available in string it will return with "value error-Substring is not found)
9. Always split method returns in list.
10. ASCII-**American Standard Code for Information Interchange.**is a character encoding standard for electronic communication. ASCII codes represent text in computers, telecommunications equipment, and other devices. Most modern character-encoding schemes are based on ASCII, although they support many additional characters.
11. For ASCII code we have function-1)ord ( ) 2)chr ( ).
12. Make_translation-For mapping of table
13. Translation-Returns translated string.

#Methods in Strings-

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string

endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits

isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found

rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

`

#Operator-

1. Operators are used to perform operations on variables and values.
2. Types of operators-

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity Operator
- Membership operators
- Bitwise operators

- [ ] **Arithmetic Operators-**

- Arithmetic operators are used with numeric values to perform common mathematical operations.
- Addition,substrction,multiplication,Division,Modulus,Expontiation,Floor division.
- Floor division-the floor division // rounds the result down to the nearest whole number.
- Modulus-To find the reminder in division.

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

- [ ] **Assignment Operator-**

- Assignment operators are used to assign values to variables.

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3

*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

- [ ] **Comparison Operator-**
  - Comparison operators are used to compare two value.

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

- [ ] **Logical Operator-**
  - Logical operators are used to combine conditional statements.

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

- [ ] **Identity Operator-**
  - Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location.

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

- Identity operator works on memory location . PF below example.

i.e.

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x

print(x is z)

# returns True because z is the same object as x

print(x is y)

# returns False because x is not the same object as y, even if they have the same content

print(x == y)

# to demonstrate the difference betweeen "is" and "==": this comparison returns True
because x is equal to y
```

- [ ] **Membership Operator-**
  - Membership operators are used to test if a sequence is presented in an object.

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

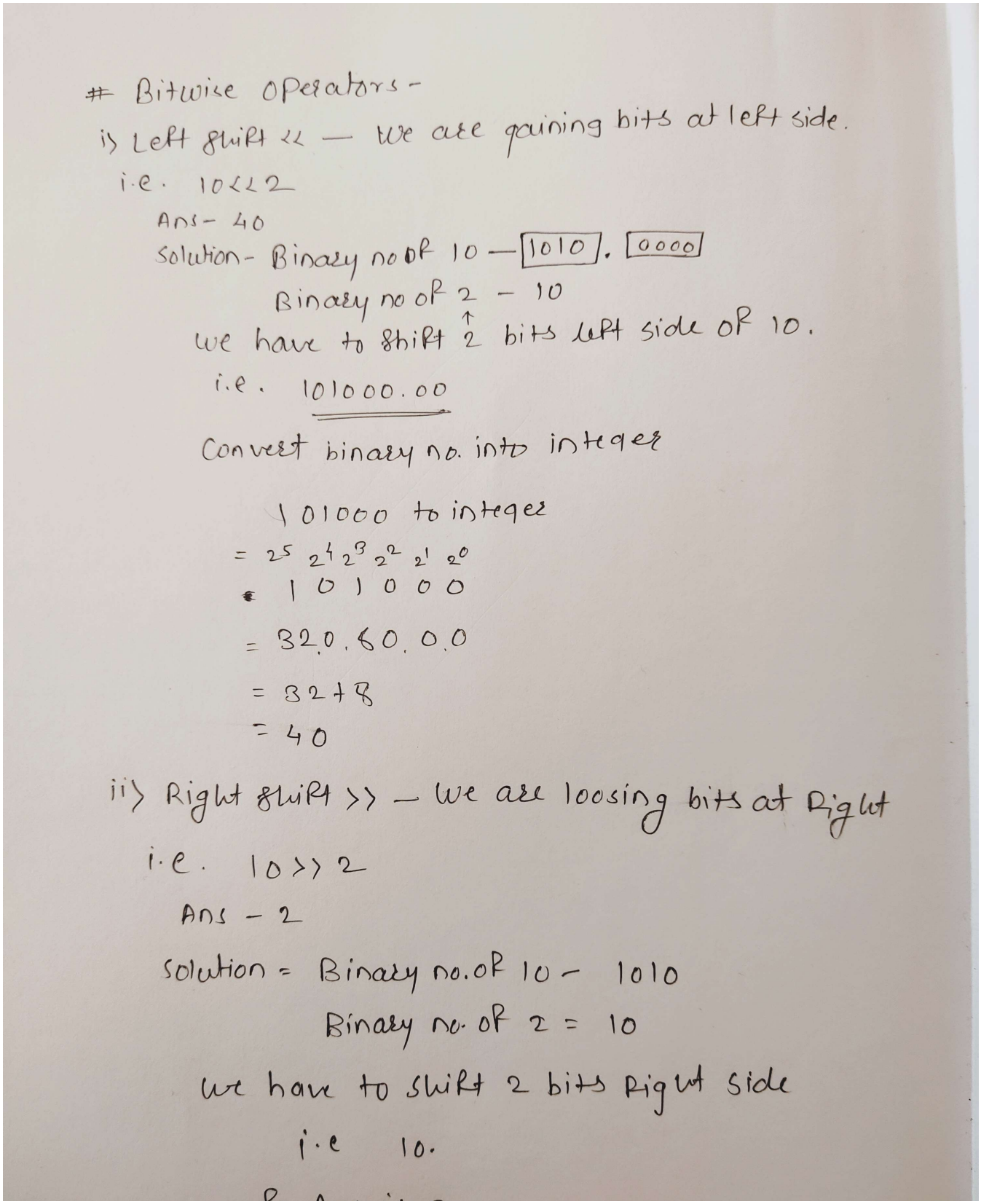
---

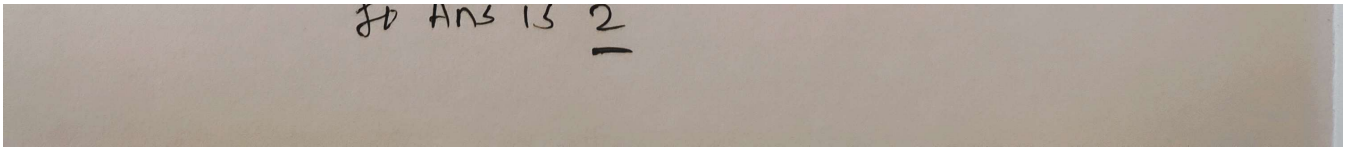
- [ ] **Bitwise Operator-**
  - Bitwise operators are used to compare (binary) numbers.
  - **And(&)**-If you have two conditions and both are true then you only get true output.
  - **Or(|)**-If you have two conditions and only one of them are true then you get true output.
  - **XOR(^)**-If you have two condition,and both are different then output will 1 : and if both conditions are same then output will o.
  - **Left Shift(<<)**-We are gaining bits at left side. for e.g- 10<<2 ans is 40,Binary no of 10 is -1010 & Binary no of 2 is-10 ,as per concept we are gaining 2 bits,so binary no 10 will -101000.

1

- Right Shift(>>)- We loosing bits at right side.

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off





#Control Statements-

-While writing code in any language, you will have to control the flow of your program. This is generally the case when there is decision making involved - you will want to execute a certain line of codes if a condition is satisfied, and a different set of code incase it is not. In Python, you have the **if**, **elif** and the **else** statements for this purpose.

- [ ] With the help of Control statements ,we can change the control/flow of program.
- [ ] With the help of logical operators i.e and ,or we can perform various task .
- [ ] Types of control statements-
  - Simple " if "
  - if-else
  - if else ladder
  - nested if else
- [ ] Python supports the usual logical conditions from mathematics:
  - Equals: a == b
  - Not Equals: a != b
  - Less than: a < b
  - Less than or equal to: a <= b
  - Greater than: a > b
  - Greater than or equal to: a >= b
- [ ] These conditions can be used in several ways, most commonly in "if statements" and loops.
- [ ] **Indentation**-Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

#Python "if" Statement-

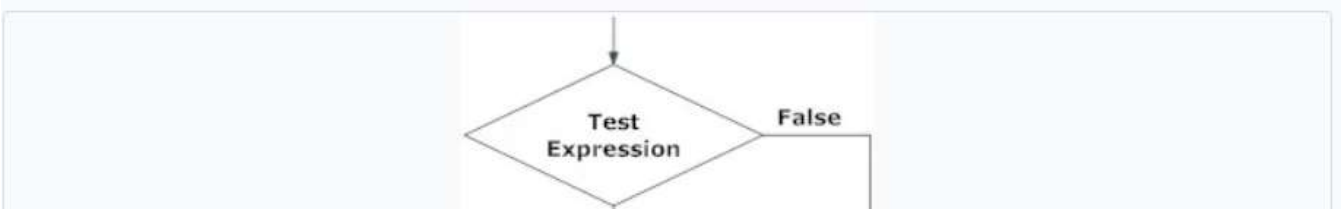
```
if test expression:
    statement(s)
```

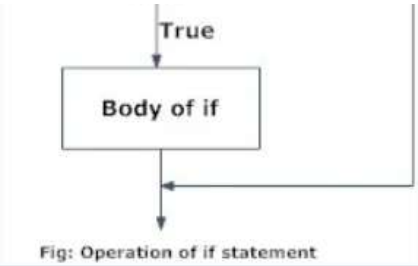
-Here, the program evaluates the test expression and will execute statement(s) only if the test expression is True.

-If the test expression is False, the statement(s) is not executed.

-In Python, the body of the if statement is indicated by the indentation. The body starts with an indentation and the first unindented line marks the end.

-Python interprets non-zero values as True. None and 0 are interpreted as False.



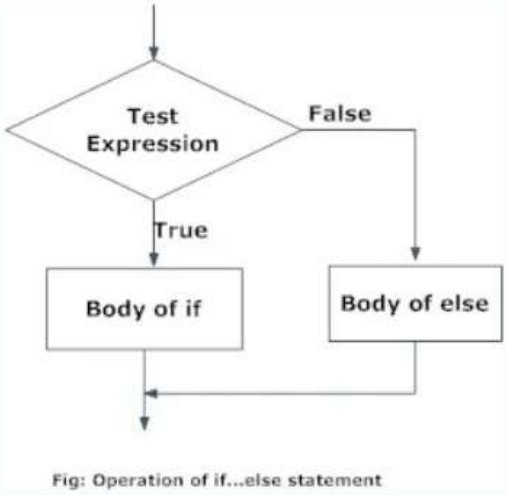


Flowchart of if statement in Python programming

#Python "if...else" Statement-

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

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



Flowchart of if...else statement in Python

#Python "if...elif...else statement-

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

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

The if block can have only one else block. But it can have multiple elif blocks.

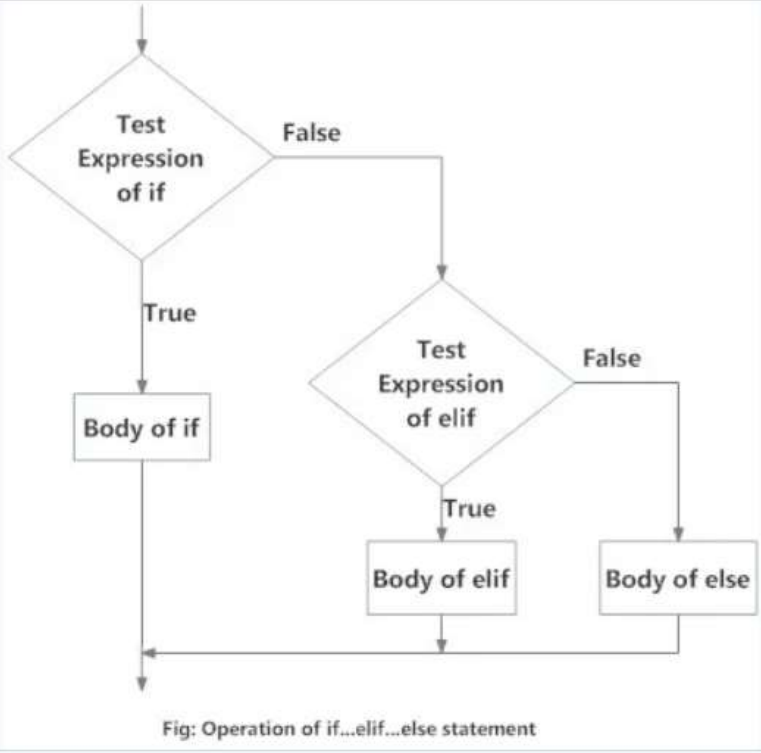


Fig: Operation of if...elif...else statement

Flowchart of if...elif....else statement in Python

#Python Nested "if" statement-

-We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming.

-Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. They can get confusing, so they must be avoided unless necessary.

#While loop and For loop-

-*Loops are important in Python or in any other programming language as they help you to execute a block of code repeatedly.* You will often come face to face with situations where you would need to use a piece of code over and over but you don't want to write the same line of code multiple times.

- [ ] **While loop-**
  - The while loop is one of the first loops that you'll probably encounter when you're starting to learn how to program. It is arguably also one of the most intuitive ones to understand: if you think of the name of this loop, you will quickly understand that the word "while" has got to do something with "interval" or a "period of time". As you already know by now, *the word "loop" refers to a piece of code that you execute repeatedly.*
  - *A while loop is a programming concept that, when it's implemented, executes a piece of code over and over again while a given condition still holds true.*
  - three components that you need to construct the while loop in Python:
    - The 'while' keyword.
    - A condition that translates to either 'True' or 'False'.
    - A block of code that you want to execute repeatedly
    - And last one 'incremental condition'.

- In [32]:
```
p=2
while p<10:
    if p%2==0:
        print("The number "+ str(p)+ " is even")
    else:
        print("The number "+ str(p)+ " is odd")
    p+=1
```

```
The number 2 is even
The number 3 is odd
The number 4 is even
The number 5 is odd
The number 6 is even
The number 7 is odd
The number 8 is even
The number 9 is odd
```

- [ ] **For Loop-**

- *The "for" component in "for loop" refers to something that you do for a certain number of times.*
- A for loop is a programming concept that, when it's implemented, executes a piece of code over
- and over again "for" a certain number of times, based on a sequence.
- following pieces that you need in order to create a for loop:
  - The 'for' keyword
  - A variable
  - The 'in' keyword
  - The 'range()' function, which is an built-in function in the Python library to create a sequence of numbers
  - The code that you want to execute repeatedly
- :
```
#For loop
#Find out valid and Invalid gmail address from below list
Email_Address=['sggije@gmail.com','abhimanyudevadhe625@gmail.com','fjeifj@rediffmail.com',]
Valid_Gmail=[]
Invalid_Gmail=[]
for i in Email_Address:
    if i.endswith('@gmail.com'):
        Valid_Gmail.append(i)
    else:
        Invalid_Gmail.append(i)
```

- :
```
Valid_Gmail
```

- :
```
['sggije@gmail.com', 'abhimanyudevadhe625@gmail.com']
```

- :
```
Invalid_Gmail
```

- :
```
['fjeifj@rediffmail.com']
```

- **Note** that the range() function's count starts from 0 and not from 1. The count should be like 0,1,2 and not 1,2,3. That's how number counting in a computer's memory works. So, while designing a for loop, always keep in mind that you have to consider the count of range from 0 and not from 1.
- *Difference Between While and For loop*-

1. In contrast to the while loop, there isn't any condition actively involved - you just execute a

piece of code repeatedly for a number of times. In other words, while the while loop keeps on executing the block of code contained within it only till the condition is True, the for loop executes the code contained within it only for a specific number of times. This "number of times" is determined by a sequence or an ordered list of things.

2. A for loop is faster than a while loop,execution time is more in While loop.

Nested Loop-

- Python programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax:

```
for iterator_var in sequence:
    for iterator_var in sequence:
        statements(s)
        statements(s)
```

- The syntax for a nested while loop statement in Python programming language is as follows:

```
while expression:
    while expression:
        statement(s)
        statement(s)
```

- A final note on loop nesting is that we can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

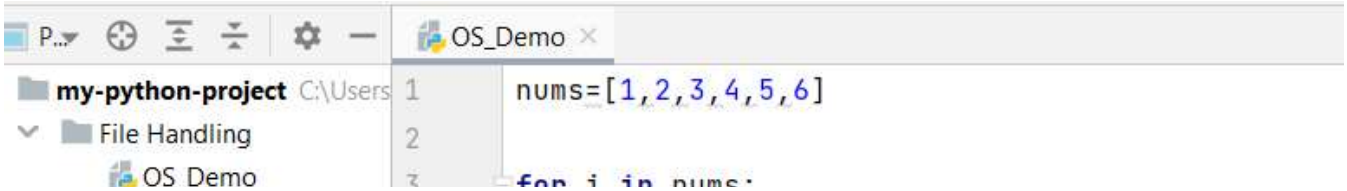
```
In [1]: #Nested Loop

        from __future__ import print_function
        for i in range(1, 5):
            for j in range(i):
                print(i, end=' ')
            print()
```

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

#Loop Control statements-

-Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.

- **Continue Statements**-It returns the control to the beginning of the loop.Control statement only ignored the specific values.



- **Break Statements-**It brings control out of the loop.

```
In [44]: #Break
```

```
In [92]: p = [1,2,3,4,5,13.33,6,7,8,9,100,100.12324,'123','python']
```

```
In [78]: for i in p:
             if i==5:
                 break
             else:
                 print(i)
```

```
1
2
3
4
```

- **Pass Statement-***We use pass statement to write empty loops*. Pass is also used for empty control statement, function and classes.

```
# An empty loop
for letter in 'geeksforgeeks':
    pass
print 'Last Letter :', letter
```

Output:

```
Last Letter : s
```

#Function-

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.
- A function is some instructions packaged together that perform a specific task .

Creating a Function-

In Python a function is defined using the **'def** 'keyword:

Example-

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

Calling a Function-

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

Example-

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

Arguments-

- Information can be passed into functions as arguments.
- Arguments are specified after the function name, inside the parentheses. We can add as many arguments as we want, just we have to separate them with a comma.
- The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
def my_function(fname):
    print(fname + " Refsnes")
my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Parameters or Arguments?

1. The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

2. From a function's perspective:

- A parameter is the variable listed inside the parentheses in the function definition.
- An argument is the value that is sent to the function when it is called.

Number of Arguments-

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Example

This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):
    print(fname + " " + lname)
my_function("Emil", "Refsnes")
```

If you try to call the function with 1 or 3 arguments, you will get an error:

Example

This function expects 2 arguments, but gets only 1:

```
def my_function(fname, lname):
    print(fname + " " + lname)my_function("Emil")
```

#Methods of Writing Arguments and Return Values-

- ***No Arguments and No Return Values-***

Example-

```
In [22]: #No arguments and No return values
         def add():
             x=int(input('Enter the first no: '))
             y=int(input('Enter the Seconf no: '))
             print(x+y)
         add()
```

```
Enter the first no: 25
Enter the Seconf no: 24
49
```

- ***With Arguments and No Return Values-***

Example-

```
In [25]: #With arguments no return values-
         def add(x,y):
             print(x+y)
         add(50,30)
```

```
80
```

- ***With Arguments and With Return Values-***

Example-

```
In [31]: #With arguments and with return values
         def add(x,y):
             z=x+y
             return print('Addition of X and Y is: ',z)
         add(25,69)
```

```
Addition of X and Y is:  94
```

- ***No Arguments with return values-***

Example-

```
In [40]: #No arguments with return Values-
         def add():
             X=int(input('Enter the first no: '))
             Y=int(input('Enter the Second no: '))
             z=X+Y
             return print('Addition of X and Y is:',z)
         add()
```

```
Enter the first no: 125
Enter the Second no: 356
Addition of X and Y is: 481
```

Types Of Arguments-

- [ ] **Positional Arguments-**
  - Arguments sequence and value which is passed through should be in same manner/position.
  - **Example-**

```
In [54]: #1.Positional Arguments-
         def Rental_Details(Name,Age,Address,Mo_No):
             print('Enter Your Name: ',Name)
             print('Enter age: ',Age)
             print('Enter Permanent Address: ',Address)
             print('Enter Mo.No: ',Mo_No)
         Rental_Details('Abhimanyu Vasantrao Devadhe',24,'Pune',7558444643)
```

```
Enter Your Name:  Abhimanyu Vasantrao Devadhe
Enter age:  24
Enter Permanent Address:  Pune
Enter Mo.No:  7558444643
```

- [ ] **Arbitrary Arguments/Variable length 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-1)**

```
In [1]: #Arbitary/Variable length Arguments-
        def temp(*args):
            print("*"*50)
            print(args)
            print("*"*50)
```

```
In [3]: temp(1,2,3,5,8,9,10)
```

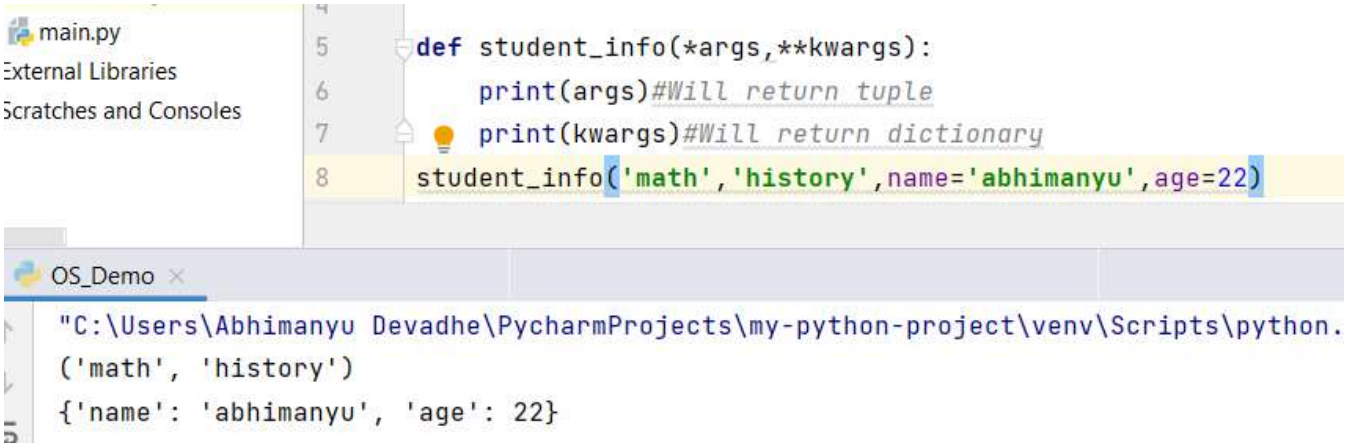
```
**************************************************
(1, 2, 3, 5, 8, 9, 10)
**************************************************
```

- If the number of arguments is unknown, add a * before the parameter name:
- **Example-2)**

```
In [5]: def my_function(*kids):
            print("The youngest child is " + kids[1])
        my_function("Emil", "Tobias", "Linus")
```

```
The youngest child is Tobias
```

- **Example-3)Combine example of variable length arguments and Keyword variable length Arguments,**



Keyword Arguments-

- You can also send arguments with the *key* = *value* syntax.
- This way the order of the arguments does not matter.

Example-

```
In [6]: #Keyword Arguments-
        def emp(name,address,age,aadhar):
            print('Name of employee: ',name)
            print('Address of employee: ',address)
            print('age of employee: ',age)
            print('Aadhar no of Employee: ',aadhar)
        emp(name='abhimanyu',age=25,aadhar=1564985,address='Pune')
```

```
Name of employee:  abhimanyu
Address of employee:  Pune
age of employee:  25
Aadhar no of Employee:  1564985
```

Arbitrary /Variable Length 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:

In [3]:
```
#Variable lenght keyword Arguments-
def my_function(**kid):
    print("His last name is " + kid["lname"])
my_function(fname = "Tobias", lname = "Refsnes")
```

```
His last name is Refsnes
```

Default Parameter Value-

- The following example shows how to use a default parameter value.
- If we call the function without argument, it uses the default value:
- **Example-1)**

In [59]:
```
#2.Default Arguments-
def My_Country(country='Norway'):
    print('I am from:',country)
My_Country('India')
My_Country('Australia')
My_Country('Sweaden')
My_Country()
```

```
I am from: India
I am from: Australia
I am from: Sweaden
I am from: Norway
```

- **Example-2)**

In [71]:
```
def emp(Name,Age,Address,Canteen='Not Available'):
    print('Enter your Full Name:',Name)
    print('Enter Your Age:',Age)
    print('Enter Your Address:',Address)
    print('Enter Canteen Facility:',Canteen)
emp('abhimanyu',25,'Pune')
```

```
Enter your Full Name: abhimanyu
Enter Your Age: 25
```

```
Enter Your Age: 25
Enter Your Address: Pune
Enter Canteen Facility: Not Available
```

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)
```

Return Values-

- To let a function return a value, use the return statement:
- **Example**

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

The pass Statement-

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

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

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.
- The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.
- 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).

- To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.
- **Example**

  Recursion Example-

```
In [19]: 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(5)
```

```
1
3
6
10
15
```

```
Out[19]: 15
```

#Lambda-

- A lambda function is a small anonymous function.
- *A lambda function can take any number of arguments, but can only have one expression.*
- Use lambda functions when an anonymous function is required for a short period of time.
- ***Lambda function used as Just In Time or Temporary function***.
- **Syntax-** lambda inputs/argument list : output expression
- The power of lambda is better shown when you use them as an anonymous function inside another function.

```
In [1]: s = lambda x,y:x*y
```

```
In [2]: s(10,30)
```

```
Out[2]: 300
```

```
In [3]: #X2+2xy+y2
        x = lambda x,y:x*x+2*x*y+y*y
```

```
In [4]: x(2,4)
```

```
Out[4]: 36
```

```
In [5]: x = lambda a,b:a if a>b else b
```

```
In [6]: x(10,50)
```

```
Out[6]: 50
```

#Mapping-

- Map takes function as argument and apply on itterables or sequence.
- Example-

```
In [19]: def f1(n):
             return n*n

In [20]: f1(3)
Out[20]: 9

In [21]: p = [3,4,8,37,29,20]

In [23]: tuple(map(f1,p))
Out[23]: (9, 16, 64, 1369, 841, 400)

In [25]: list(map(lambda x:x*x,p))
Out[25]: [9, 16, 64, 1369, 841, 400]
```

#Filter-

```
In [34]: def even(x):
             if x%2==0:
                 return True
             else:
                 return False

In [35]: even(45)
Out[35]: False

In [36]: p
Out[36]: ['java', 'python', 'c', 'c++', 'Hadoop']

In [37]: p = [4,67,24,98,35,68]

In [38]: list(filter(even,p))
Out[38]: [4, 24, 98, 68]
```

```
In [40]: list(filter(lambda x:x%2==0,p))
Out[40]: [4, 24, 98, 68]
```

#Reduce-

```
In [43]: p
Out[43]: [4, 67, 24, 98, 35, 68]

In [44]: sum(p)
Out[44]: 296

In [45]: min(p)
Out[45]: 4

In [46]: max(p)
Out[46]: 98

In [53]: from functools import reduce

In [54]: p
Out[54]: [4, 67, 24, 98, 35, 68]

In [55]: reduce(lambda x,y:x/y,p)
Out[55]: 1.0665246908998141e-08
```

#Function Aliasing -

- In Python, aliasing **happens whenever one variable's value is assigned to another variable**, because variables are just names that store references to values.
- Same like variables **we can assign function to another variable.**

```
In [8]: def add(x,y):
            print(x+y)

In [9]: add(20,30)
        50

In [10]: s=add
```

```
In [20]: s(213,10)
         223

In [21]: A1=s

In [22]: A1(195,50)
         245

In [28]: print(id(A1),id(add),id(s))
         2852154019552 2852154019552 2852154019552
```

#Local and Global Variables in Python-

- [ ] **Global Variables-**

- **A variable declared outside of the function or in global scope** is known as a global variable. This means that a global variable can be accessed inside or outside of the function.

```
In [25]: course = 'python'
         def temp():
             global course
             course = 'java'
             print(course)
         def temp1():
             print(course)
         def m1():
             print(course)
         def m2():
             global course
             course='c++'
             print(course)

In [27]: print(temp(),m1(),m2(),temp())
         java
         java
         c++
         java
         None None None None
```

- **Global Keyword-**

Global Keyword use for,

1. Accessing global variable.
2. Change value of Global variable.
3. Make variable as global

3. Make variable as global.

```
In [35]: def s(t1,t2,t3):
             global newt1
             newt1=t1
             global newt2
             newt2=t2
             global newt3
             newt3=t3
             print(t1),print(t2),print(t3)
```

```
In [36]: s(1,2,3)
```

```
1
2
3
```

- [ ] **Local Variables-**

- A variable declared inside the function's body or in the local scope is known as a local variable.

```
In [37]: def foo():
             y = "local"
             print(y)

         foo()
```

```
local
```

- Example for Combine Global and Local Variable-

```
In [44]: x='Global'
         def f1():
             global x
             x= x * 2
             y='Local'
             print(x)
             print(y)
         f1()
```

```
GlobalGlobal
Local
```

#Function Decorators-

- Decorators are a very powerful and useful tool in Python since it allows programmers to modify the behaviour of function or class.
- ***Decorators are used to modify the behaviour of function or class. In Decorators,***

functions are taken as the argument into another function and then called inside the wrapper function.

- Decorators allow us to wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it.
- ***Decorators are used for File handling,Database connection and API call.***
- **functions are first class objects that mean that functions in Python can be used or passed as arguments.**
- Properties of first class object-

1. A function is an instance of the Object type.
2. You can store the function in a variable.
3. You can pass the function as a parameter to another function.
4. You can return the function from a function.
5. You can store them in data structures such as hash tables, lists,.....

- **Example 1-**

In [2]:
```
def decor(fun):
    def inner():
        print('#'*15)
        fun()
        print('#'*15)
    return inner
@decor
def greet():
    print('Welcome In Programming World')
greet()
```

```
###############
Welcome In Programming World
###############
```

- **Example 2**

In [6]:
```
# defining a decorator
def hello_decorator(func):

# inner1 is a Wrapper function in
# which the argument is called

# inner function can access the outer local
# functions like in this case "func"
    def inner1():
        print("Hello, this is before function execution")

        # calling the actual function now
        # inside the wrapper function.
        func()

        print("This is after function execution")

    return inner1

#using @hello_decorator for call calling decrator
@hello_decorator
# defining a function, to be called inside wrapper
def function_to_be_used():
```

```
    print("This is inside the function !!")

# calling the function
function_to_be_used()
```

```
Hello, this is before function execution
This is inside the function !!
This is after function execution
```

#Iterator and Generator-

- [ ] **Iterator-**

- An iterator is an object which contains a countable number of values and it is used to iterate over iterable objects like list, tuples, sets, etc. Iterators are implemented using a class and a local variable for iterating is not required here, It follows lazy evaluation where the evaluation of the expression will be on hold and stored in the memory until the item is called specifically which helps us to avoid repeated evaluation.
- As lazy evaluation is implemented, it requires only 1 memory location to process the value and when we are using a large dataset then, wastage of RAM space will be reduced the need to load the entire dataset at the same time will not be there.
  Using an iterator-
  - **iter()** keyword is used to create an iterator containing an iterable object.
  - **next()** keyword is used to call the next element in the iterable object.
  - After the iterable object is completed, to use them again reassign them to the same object.

```
In [1]: iter_list=iter(['java','python','c','c++'])
        print(next(iter_list))
        print(next(iter_list))
        print(next(iter_list))
        print(next(iter_list))
```

```
java
python
c
c++
```

- [ ] **Generator-**

- It is another way of creating iterators in a simple way where it uses the keyword “yield” instead of returning it in a defined function.
- Generators are implemented using a function. Just as iterators, generators also follow lazy evaluation. Here, the yield function returns the data without affecting or exiting the function.
- It will return a sequence of data in an iterable format where we need to iterate over the sequence to use the data as they won’t store the entire sequence in the memory.
-

In [18]:
```
def square_num(n):
    for i in range(1,n+1):
        yield i*i

a=square_num(3)


print("The square of numbers 1,2,3 are : ")
print(next(a))
print(next(a))
print(next(a))
```

```
The square of numbers 1,2,3 are :
1
4
9
```

- Difference Between Iterator and Generator-

Iterator	Generator
Class is used to implement an iterator	Function is used to implement a generator.
Local Variables aren't used here.	All the local variables before the yield function are stored.
Iterators are used mostly to iterate or convert other objects to an iterator using iter() function.	Generators are mostly used in loops to generate an iterator by returning all the values in the loop without affecting the iteration of the loop
Iterator uses iter() and next() functions	Generator uses yield keyword
Every iterator is not a generator	Every generator is an iterator

---

#Code Optimization Techniques-

List Comprehensions-

- In Python, a great syntactic construct that is computationally more efficient for creating lists than a traditional loop is list comprehensions. So if you need to, say, have a binary feature vector for a list of numbers as data points where all negative numbers will be assigned 0 and the rest will be assigned 1, instead of:

In [11]:
```
#Simple code without using List Comprehension techniques-
input_list=[1,-2,3,-4,4,5,6]
output_list=[]
for x in input_list:
    if x>=1:
```

```
        output_list.append(1)
    else:
        output_list.append(0)
output_list
```

Out[11]: [1, 0, 1, 0, 1, 1, 1]

In [13]:
```
#using List Comprehension techniques-
output_list=[1 if x>=1 else 0 for x in input_list ]
output_list
```

Out[13]: [1, 0, 1, 0, 1, 1, 1]

- [ ] **Avoid for-loops and list comprehensions where possible-**

- In fact, in the previous example, if you were creating a vector with just one initialization value, instead of using inefficient for-loops and even list comprehensions, you could do something like that:

In [20]:
```
my_list1=[1,3,6,5,8]
my_list2 = [0] * len(my_list1)
my_list2
```

Out[20]: [0, 0, 0, 0, 0]

- [ ] **Avoid unnecessary functions-**

- A good example of an approach that can help shave off a significant amount of the runtime complexity but requires a lot of careful thought in terms of trade-offs is function calls. While you do want the nice abstraction, extensibility, and re-usability that functions provide, you might not want to have a function for every single thing, because function calls are quite expensive in Python . So sometimes, you might want to sacrifice, for example, writing a getter and/or a setter. On the other hand, fewer functions make the code less testable. So the final decision really depends on the specifics of your application.
- ***Tuple comprehension is not possible in python,if we try for same it will create Generator.***

In [57]:
```
p = tuple(range(10))
```

In [58]:
```
p
```

Out[58]: (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

In [59]:
```
k = (i for i in p)
```

In [60]:
```
type(k)
```

```
In [60]: type(k)
Out[60]: generator

In [61]: k = [ i for i in p ]

In [62]: type(k)
Out[62]: list
```

- ***Dictionary Comprehensions is also possible in python.***

```
In [35]: p= {0:0,1:1,2:2,3:3,4:4}

In [39]: P= {i:i for i in range(5)}

In [40]: P
Out[40]: {0: 0, 1: 1, 2: 2, 3: 3, 4: 4}
```

#File Handling-

- Python supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files.
- Python treats file differently as text or binary and this is important.

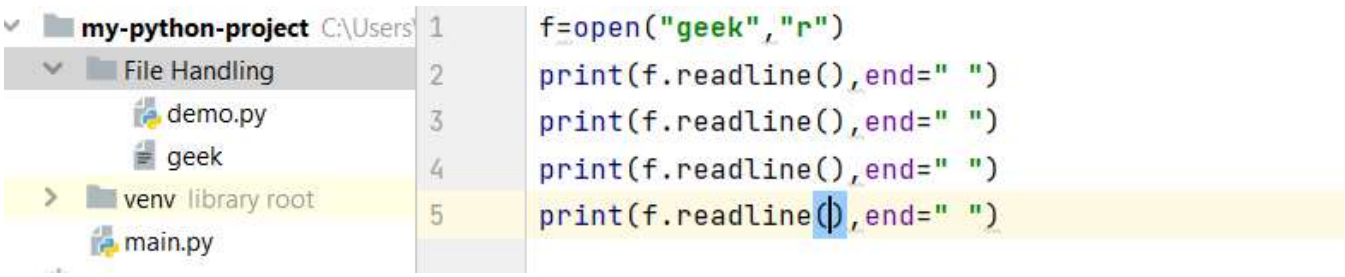
☐ **Working of open() function-**

- We use **open ()** function in Python to open a file in read or write mode. Open ( ) will return a file object.
- To return a file object we use **open()** function along with two arguments, that accepts file name and the mode, whether to read or write.
- So, the syntax being: ***open(filename, mode)***. There are three kinds of mode, that Python provides and how files can be opened:

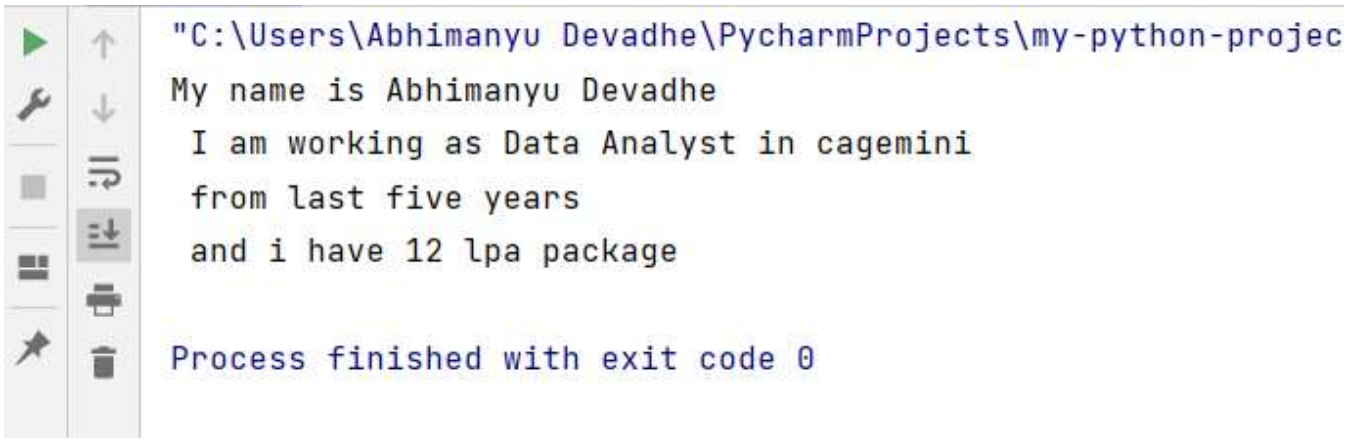
1. " **r** ", for reading.
2. " **w** ", for writing.
3. " **a** ", for appending.
4. " **r+** ", for both reading and writing.

☐ **Read Mode -**

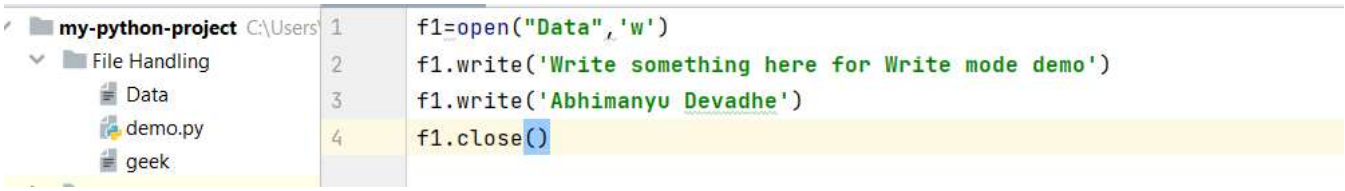
- There is more than one way to read a file in Python. If you need to extract a string that contains all characters in the file then we can use **file.read()**.
- If file is present it will read the data,if not it will return error i.e file not found.
- *One must keep in mind that the mode argument is not mandatory. If not passed, then Python will assume it to be " r " by default.*
- **Example-**



- **Output-**

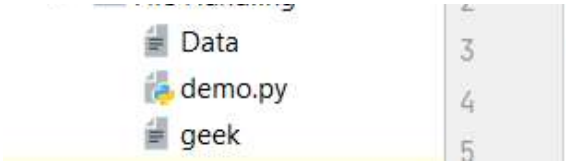


- [ ] **Write M0de-**
  - Write mode to manipulate the file we use write mode.
  - In write mode data will over-write and old data will delete.
  - The ***close() command terminates all the resources*** in use and frees the system of this particular program.

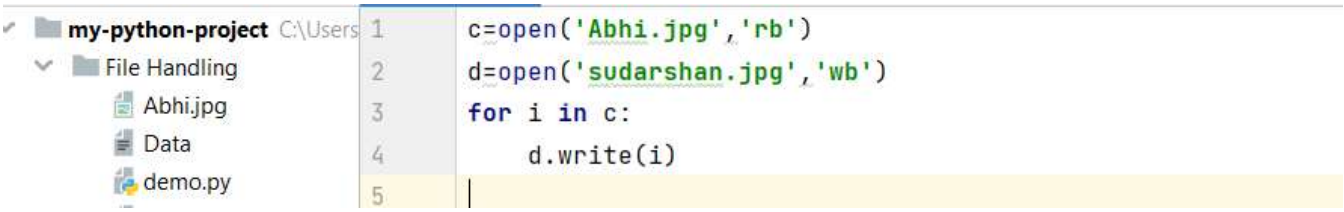


- [ ] **Append Mode-**
  - Data will added at the end of text line-

- ***Copy of data from one file to another file-***



- ***Copy image from one jpg to another jpg***

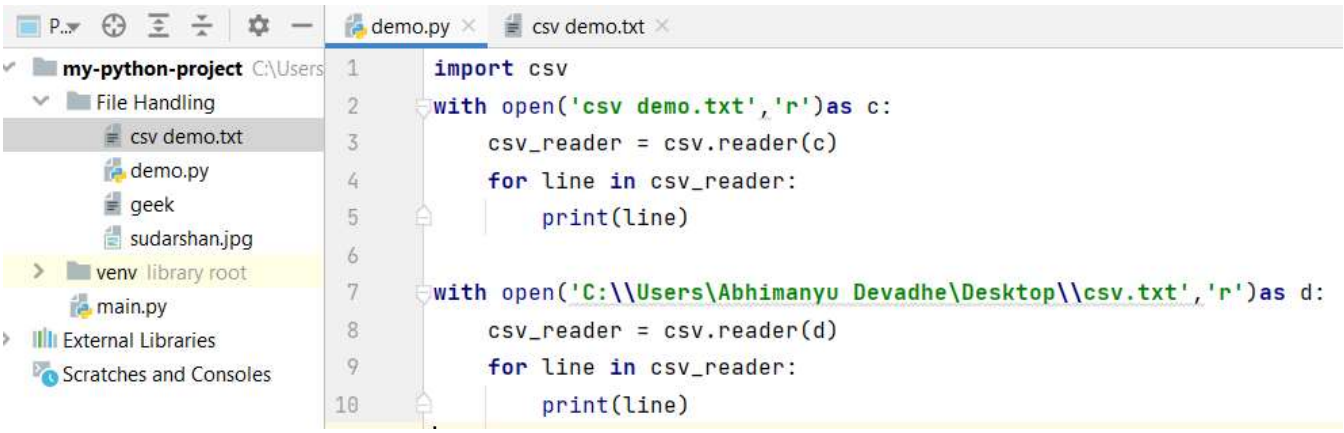


#CSV in Python-

- "Comma Separated Values" format is the most common import and export format for spreadsheets and databases.*Values are separated with comma's so called as Comma separated value and what separates two values is called delimiter.*
- The csv module implements classes to read and write tabular data in CSV format.
- It allows programmers to say, “write this data in the format preferred by Excel,” or “read data from this file which was generated by Excel,” without knowing the precise details of the CSV format used by Excel. Programmers can also describe the CSV formats understood by other applications or define their own special-purpose CSV formats.
- The CSV Module Defines following functions-

csv.reader-

- Syntax-csv.reader(*csvfile*, *dialect='excel'*, ***fmtparams*)
- Return a reader object which will iterate over lines in the given *csvfile*. *csvfile* can be any object which supports the iterator protocol and returns a string each time its __next__() method is called — file objects and list objects are both suitable.
- ***Each row read from the csv file is returned as a list of strings.***
- Example-program for file in same dictionary and file at another dictionary.



---

csv.writer-

- csv.**writer**(*csvfile*, *dialect='excel'*, ***fmtparams*).
- Return a writer object responsible for converting the user's data into delimited strings on the given file-like object. *csvfile* can be any object with a write() method. If *csvfile* is a file object, it should be opened with newline=''

---

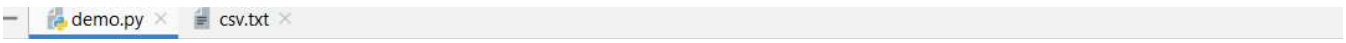
csv.DictReader-

- Syntax-csv.**DictReader**(*f*, *fieldnames=None*, *restkey=None*, *restval=None*, *dialect='excel'*, **args*, ***kwds*)
- Create an object that operates like a regular reader but maps the information in each row to a dict whose keys are given by the optional *fieldnames* parameter.
- The *fieldnames* parameter is a sequence. If *fieldnames* is omitted, the values in the first row of file *f* will be used as the fieldnames. Regardless of how the fieldnames are determined, the dictionary preserves their original ordering.
- We can fetch desired values with specified key.

---

csv.DictWriter-

- Syntax-csv.**DictWriter**(*f*, *fieldnames*, *restval=''*, *extrasaction='raise'*, *dialect='excel'*, **args*, ***kwds*)
- Create an object which operates like a regular writer but maps dictionaries onto output rows. The *fieldnames* parameter is a sequence of keys that identify the order in which values in the dictionary passed to the writerow() method are written to file *f*.
- The optional *restval* parameter specifies the value to be written if the dictionary is missing a key in *fieldnames*. If the dictionary passed to the writerow() method contains a key not found in *fieldnames*, the optional *extrasaction* parameter indicates what action to take. If it is set to 'raise', the default value, a ValueError is raised. If it is set to 'ignore', extra values in the dictionary are ignored.



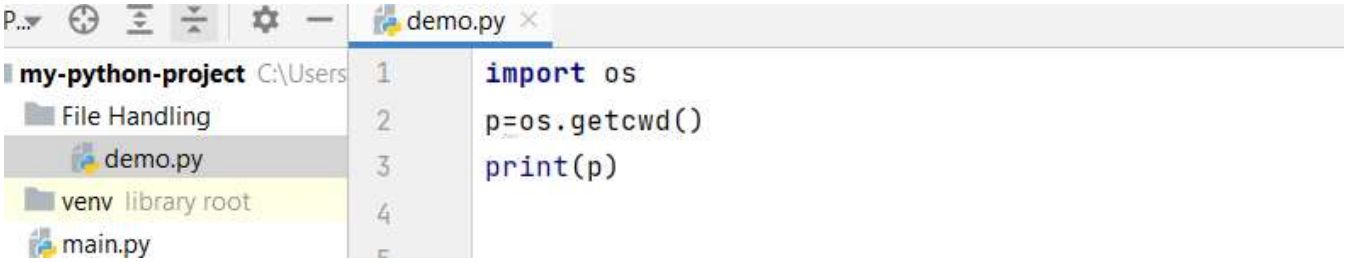
```python
import csv
with open('csv.txt','r') as c:
    csv_reader=csv.DictReader(c)
    with open('csv_demo', 'w') as d:
        fieldnames=['Year','Industry_aggregation_NZSIOC','Variable_code','Variable_name','Value']
        csv_writer = csv.DictWriter(d,fieldnames=fieldnames)
        csv_writer.writeheader()
        for line in csv_reader:
            csv_writer.writerow(line)
```

#OS Module in Python-

- The OS module in Python provides functions for interacting with the operating system. OS comes under Python's standard utility modules.
- This module provides a portable way of using operating system-dependent functionality. The *os* and *os.path* modules include many functions to interact with the file system.

- [ ] **Attributes or Methods in OS Module-**
- [ ] **Handling the Current Working Dictionary-**

1. Current Working Directory(CWD) as a folder, where the Python is operating.
2. Whenever the files are called only by their name, Python assumes that it starts in the CWD which means that name-only reference will be successful only if the file is in the Python's CWD.
3. *Note: The folder where the Python script is running is known as the Current Directory. This is not the path where the Python script is located.*



- [ ] **Changing the Current Working Dictionary-**

1. To change the current working directory(CWD) os.chdir() method is used. This method changes the CWD to a specified path. It only takes a single argument as a new directory path.
2. *Note-The current working directory is the folder in which the Python script is operating.*

- [ ] **Creating Directory-**

- ***os.mkdir( )-***

1. os.mkdir() method in Python is used to create a directory named path with the specified numeric mode.
2. This method raises FileExistsError if the directory to be created already exists.
3. os.mkdir is used for create directory ;can not create intermediate dict/files.

- ***os.makedirs( )-***

1. os.makedirs() method in Python is used to create a directory recursively.
2. That means while making leaf directory if any intermediate-level directory is missing, os.makedirs() method will create them all.
3. os.makedirs() is used for create directory ;can create intermediate dict/file.

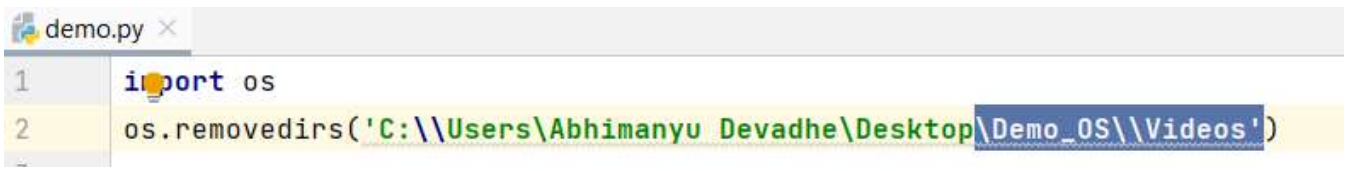
- [ ] **Delete Directory-**

- ***os.rmdir( )-***

1. We can remove specified directory but we can not remove intermediate directories.
2. os.rmdir() method in Python is used to remove or delete an empty directory. OSError will be raised if the specified path is not an empty directory.
3. Recommanded for delete dictionaries.
4. Example- Highlighted Direct/folder will delete.

- ***os.removedirs( )-***

1. We can remove Directory and sub-Directory with the help of os.removedirs( ).
2. Not strongly recommanded for deletion purpose due to whole directory can be removed.



```
3
4
```

- ***os.walk ( )-***

1. Traverse the directory tree and print all of the directories and files
2. So walk is a Generator that yields a couple of three values as walking the directory tree ,each directory that it sees it yields the directory path the direct within that path .
3. Example-With the help of os.walk module we can traverse all directories and files at specified path.

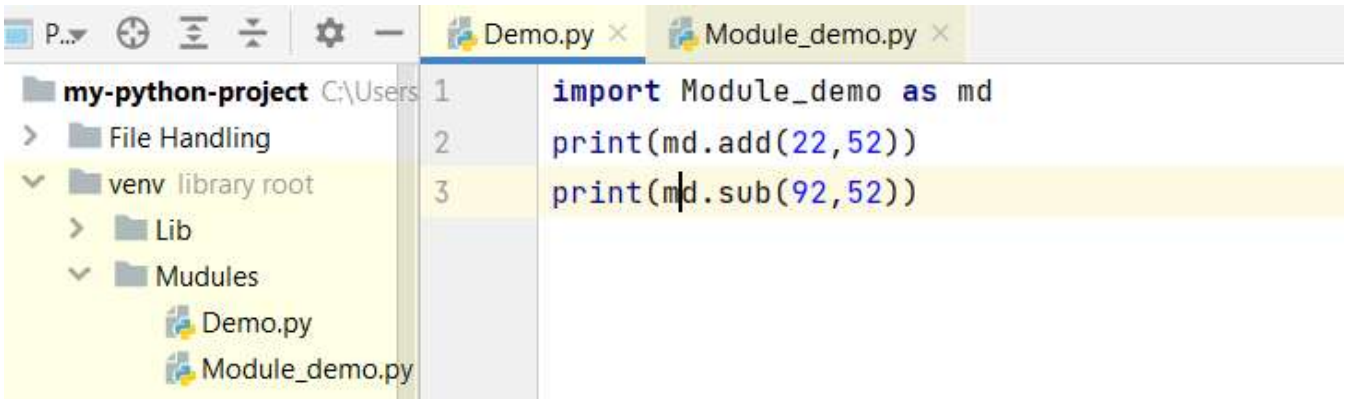
```
sers 1  import os
     2  for dirpath,dirnames,filenames in os.walk('C:\\Users\Abhimanyu Devadhe\\Desktop'):
     3      print('Current Dir path: ',dirpath)
     4      print('Directories: ',dirnames)
     5      print('Files_Names: ',filenames)
     6      print()
```

#Python Modules-

- A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code.
- Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

- [ ] ***Import Module in Python – Import statement-***

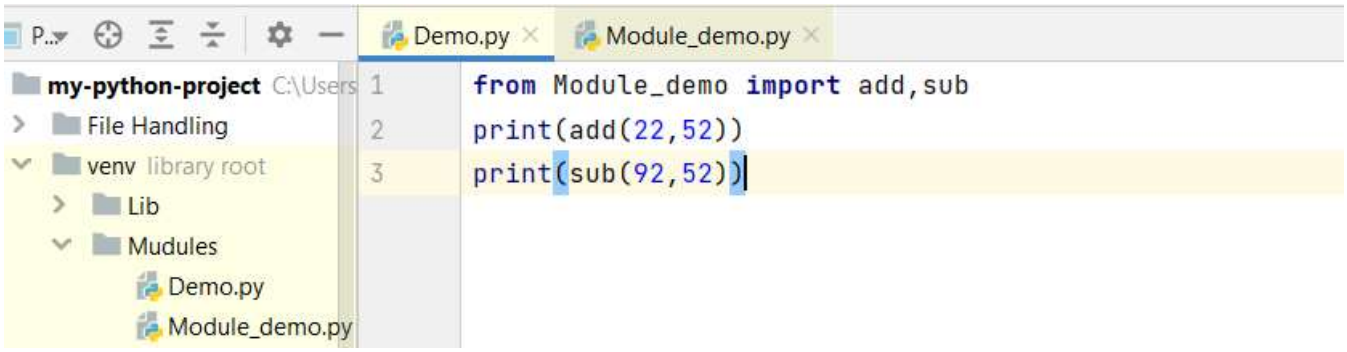
- We can import the functions, classes defined in a module to another module using the **import statement** in some other Python source file.
- When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches for importing a module. For example, to import the module Module_demo.py, we need to put the following command at the top of the script.



- [ ] ***The from import Statement*** -

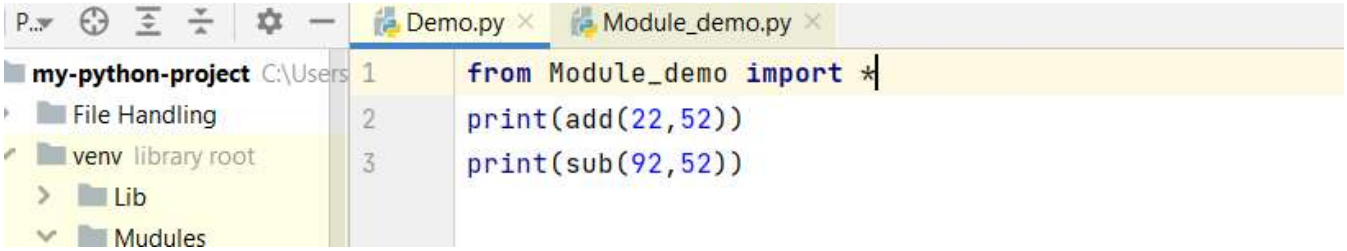
- Python's *from* statement lets you import specific attributes from a module without importing

- Python's *from* statement lets you import specific attributes from a module without importing the module as a whole like above example.



- [ ] ***Import all Names – From import * Statement***

- The * symbol used with the from import statement is used to import all the names from a module to a current namespace.



- [ ] ***Locating Modules-***

- Whenever a module is imported in Python the interpreter looks for several locations. First, it will check for the built-in module, if not found then it looks for a list of directories defined in the sys.path.
- ***Python interpreter searches for the module in the following manner***

1. First, it searches for the module in the current directory.
2. If the module isn't found in the current directory, Python then searches each directory in the shell variable PYTHONPATH. The PYTHONPATH is an environment variable, consisting of a list of directories.
3. If that also fails python checks the installation-dependent list of directories configured at the time Python is installed.

```
# importing sys module
import sys

# importing sys.path
print(sys.path)
```

- [ ] ***Note-We can give path for module which is not in directory of scripts-***

- ***Example 1-With the help add path in sys.path***

- ***Example 1-With the help add path in sys.path***

- ***Example 2-With the help changing path of Environment Variable***

1. Create new environment variable from control panel\Edit environment variable.i.e give Title as PYTHONPATH and path as module location.
2. then go to Command Prompt and simply import your module.

#Python Packages-

- We usually organize our files in different folders and subfolders based on some criteria, so that they can be managed easily and efficiently. For example, we keep all our games in a Games folder and we can even subcategorize according to the genre of the game or something like this. The same analogy is followed by the Python package.
- Python Module may contain several classes, functions, variables, etc. whereas a ***Python package can contains several module. In simpler terms a package is folder that contains various modules as files.***

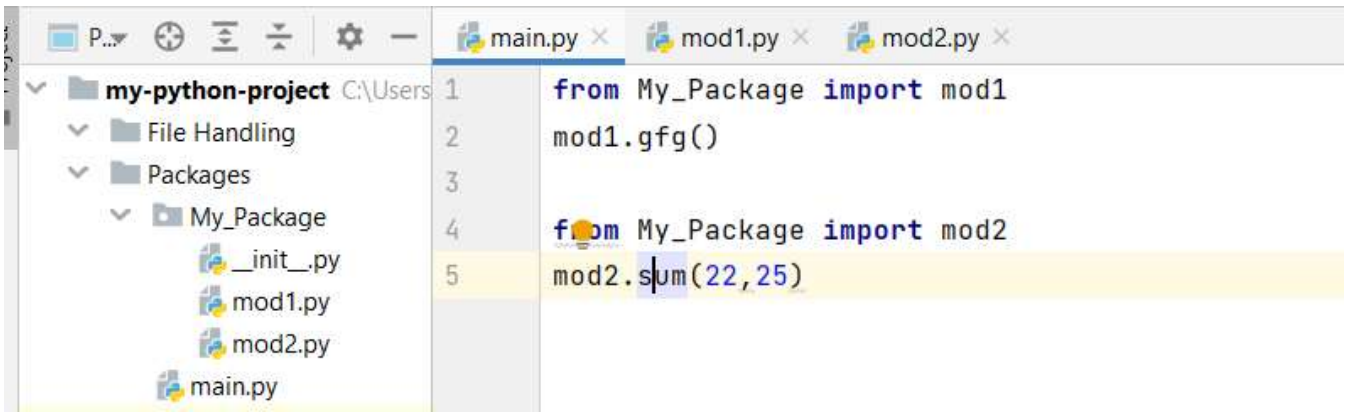
Creating Package-

- create a package named mypckg that will contain two modules mod1 and mod2. To create this module follow the below steps –

1. Create a folder named mypckg.
2. Inside this flder create an empty Python file i.e. __init__.py
3. Then create two modules mod1 and mod2 in this folder.

Understanding __init__.py-

- __init__.py helps the Python interpreter to recognise the folder as package. It also specifies the resources to be imported from the modules. If the __init__.py is empty this means that all the functions of the modules will be imported. We can also specify the functions from each module to be made available.
- For example, we can also create the __init__.py file for the above module as –

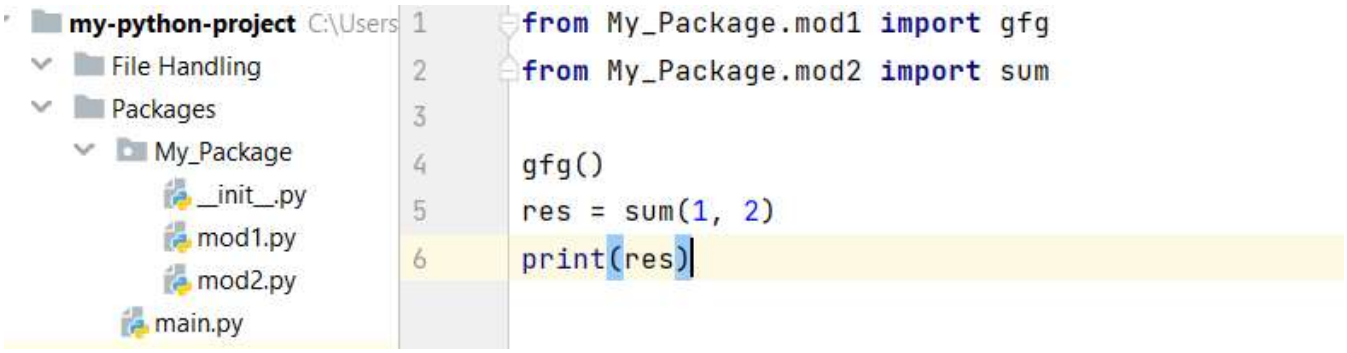


Import Modules from Package-

- We can import these modules using the from.....importstatement and the dot(.) operator.
- Syntax-

```
import package_name.module_name
```

- **Example: Import Specific function from the module-**
- ***We can also import modules from any Directory.***



#Virtual Environment In Python-

- A Virtual Environment is a python environment, that is an isolated working copy of Python which allows you to work on a specific project without affecting other projects.
- So basically it is a tool that enables multiple side-by-side installations of Python, one for each project.

- [ ] **pip list-**
  - pip list showing modules that had install during system installation of Python.

```
C:\Users\Abhimanyu Devadhe>cd desktop

C:\Users\Abhimanyu Devadhe\Desktop>pip list
Package                           Version
--------------------------------- -------------------
alabaster                         0.7.12
anaconda-client                   1.7.2
anaconda-navigator                2.0.3
anaconda-project                  0.9.1
anyio                             2.2.0
```

- [ ] **Creating Python Virtual Environment-**
  - To create new Virtual environment, simply type ***python -m venv (your project name i.e)my_project***

```
C:\Users\Abhimanyu Devadhe\Desktop>python -m venv my_project
```

- When you run -m python will search your sys path and execute that module as the main module.
- Now we have new virtual environment at desktop.

```
C:\Users\Abhimanyu Devadhe\Desktop>dir
 Volume in drive C is Windows
 Volume Serial Number is 6A64-9A68

 Directory of C:\Users\Abhimanyu Devadhe\Desktop

26-11-2021  11:49    <DIR>          .
26-11-2021  11:49    <DIR>          ..
07-08-2021  10:37            26,597 Abhi.jpg
14-11-2021  06:54    <DIR>          DataScience
22-11-2021  18:19    <DIR>          Documents
05-11-2021  10:27            76,146 EMI.PNG
26-11-2021  10:36        10,988,477 Lecture Notes.pdf
04-11-2021  06:56    <DIR>          Linux
26-11-2021  11:49    <DIR>          my_project
15-11-2021  14:49           201,038 Passbook.jpg
07-10-2021  20:17             1,282 Pycharm.lnk
24-10-2021  11:44    <DIR>          Resume
31-08-2021  18:43               893 Servicenow Links.txt
23-10-2021  12:35    <DIR>          SQL
13-10-2021  23:04    <DIR>          Videos
               6 File(s)     11,294,433 bytes
               9 Dir(s)  912,333,766,656 bytes free
```

- [ ] **Activate Virtual Environment-**

- To activate this environment simply type-
- ***(Your project name i.e)my_project\|scripts\|activate.bat***

```
C:\Users\Abhimanyu Devadhe\Desktop>my_project\scripts\activate.bat

(my_project) C:\Users\Abhimanyu Devadhe\Desktop>
```

- We can find our Virtual environment location.

```
(my_project) C:\Users\Abhimanyu Devadhe\Desktop>where python
C:\Users\Abhimanyu Devadhe\Desktop\my_project\Scripts\python.exe
C:\Users\Abhimanyu Devadhe\anaconda3\python.exe
C:\Users\Abhimanyu Devadhe\AppData\Local\Programs\Python\Python310\python.exe
C:\Users\Abhimanyu Devadhe\AppData\Local\Microsoft\WindowsApps\python.exe

(my_project) C:\Users\Abhimanyu Devadhe\Desktop>
```

- Installed packages with this environment.

```
(my_project) C:\Users\Abhimanyu Devadhe\Desktop>pip list
Package    Version
---------- -------
pip        20.2.3
setuptools 49.2.1
```

```
setuptools 49.2.1
WARNING: You are using pip version 20.2.3; however, version 21.3.1 is available.
You should consider upgrading via the 'c:\users\abhimanyu devadhe\desktop\my_project\

(my_project) C:\Users\Abhimanyu Devadhe\Desktop>
```

- [ ] **Install packages one by one-**
  - Simply type -
  - ***pip install (package name and current version that you have to install) pandas==1.3.4***

```
(my_project) C:\Users\Abhimanyu Devadhe\Desktop>pip install pandas==1.3.4
Collecting pandas==1.3.4
  Using cached pandas-1.3.4-cp38-cp38-win_amd64.whl (10.2 MB)
Collecting python-dateutil>=2.7.3
  Using cached python_dateutil-2.8.2-py2.py3-none-any.whl (247 kB)
Collecting pytz>=2017.3
  Using cached pytz-2021.3-py2.py3-none-any.whl (503 kB)
Collecting numpy>=1.17.3; platform_machine != "aarch64" and platform_machine != "arm64"
  Using cached numpy-1.21.4-cp38-cp38-win_amd64.whl (14.0 MB)
Collecting six>=1.5
  Using cached six-1.16.0-py2.py3-none-any.whl (11 kB)
Installing collected packages: six, python-dateutil, pytz, numpy, pandas
```

- [ ] **Export Packages used for Projects-**
  - ***pip freeze>requirement.txt***

```
(my_project) C:\Users\Abhimanyu Devadhe\Desktop>pip freeze>requirement.txt
```

  - requirement.txt file will create your desktop.

- [ ] **Deactivate Virtual Environment-**
  - ***deactivate***

```
(my_project) C:\Users\Abhimanyu Devadhe\Desktop>deactivate
C:\Users\Abhimanyu Devadhe\Desktop>
```

- [ ] **Delete Environment-**
  - ***rmdir (your project name i.e)my_project /s***
  - ***/s*** will delete entire directory.

```
(my_project) C:\Users\Abhimanyu Devadhe\Desktop>deactivate
C:\Users\Abhimanyu Devadhe\Desktop> rmdir my_project /s
my_project, Are you sure (Y/N)? y
```

- [ ] **Install packages with the help of requirement.txt**

- Create my_project directory at desktop and under that make venv environment.

```
C:\Users\Abhimanyu Devadhe\Desktop>mkdir my_project

C:\Users\Abhimanyu Devadhe\Desktop>python -m venv my_project\venv
```

- Activate venv environment-

```
C:\Users\Abhimanyu Devadhe\Desktop>my_project\venv\scripts\activate.bat

(venv) C:\Users\Abhimanyu Devadhe\Desktop>
```

- Install packages at a time in new created environment,requirements.txt file is at desktop.
- simply type ***pip install -r requirements.txt***

```
(venv) C:\Users\Abhimanyu Devadhe\Desktop>pip install -r requirements.txt
Collecting numpy==1.21.4
  Using cached numpy-1.21.4-cp38-cp38-win_amd64.whl (14.0 MB)
Collecting pandas==1.3.4
  Using cached pandas-1.3.4-cp38-cp38-win_amd64.whl (10.2 MB)
Collecting python-dateutil==2.8.2
  Using cached python_dateutil-2.8.2-py2.py3-none-any.whl (247 kB)
Collecting pytz==2021.3
  Using cached pytz-2021.3-py2.py3-none-any.whl (503 kB)
Collecting six==1.16.0
  Using cached six-1.16.0-py2.py3-none-any.whl (11 kB)
Installing collected packages: numpy, six, python-dateutil, pytz, pandas
```

#Object Serialization -

- Pickling is the process whereby a Python object hierarchy is converted into a byte stream (usually not human readable) to be written to a file, this is also known as Serialization. Unpickling is the reverse operation, whereby a byte stream is converted back into a working Python object hierarchy.
- Pickle is operationally simplest way to store the object. The Python Pickle module is an object-oriented way to store objects directly in a special storage format.
- ***We can convert object from python to common data transformation platform i.e json,pickle,yaml etc.***

pickle( ) -

- Pickling is the process whereby a *Python object hierarchy is converted into a byte stream/binary format (usually not human readable) to be written to a file*, this is also *known as Serialization*. Unpickling is the reverse operation, whereby a byte stream is converted back into a working Python object hierarchy.
- Pickle is operationally simplest way to store the object. The Python Pickle module is an object-oriented way to store objects directly in a special storage format.
- ***What can it do?***

1. Pickle can store and reproduce dictionaries and lists very easily.
2. Stores object attributes and restores them back to the same State.

- ***What pickle can't do?***

1. It does not save an objects code. Only it's attributes values

1. It does not save an objects code. Only it's attributes values.
2. It cannot store file handles or connection sockets.

- In short we can say, pickling is a way to store and retrieve data variables into and out from files where variables can be lists, classes, etc.
- Syntax-

```
pickle.dump(mystring, outfile, protocol)
```

- ***Methods in pickle interface-***

1. **dump()** – The dump() method serializes to an open file (file-like object).
2. **dumps()** – Serializes to a string.
3. **load()-**Deserializes from an open-like object.
4. **loads()**-Deserializes from a string.

Pickling-

- **dump( )-**The dump() method serializes to an open file (file-like object)

```
#Pickling
import pickle
d={'name':'Abhimanyu','age':24,'Salary':'45000'}

with open('mod3.py','wb') as fp:
    pickle.dump(d, fp)
```

- **dumps( )-**Serializes to a string.

main.py mod3.py

```
#Pickling
import pickle
d="""{'name':'Abhimanyu','age':24,'Salary':'45000'}"""
print(pickle.dumps(d))
```

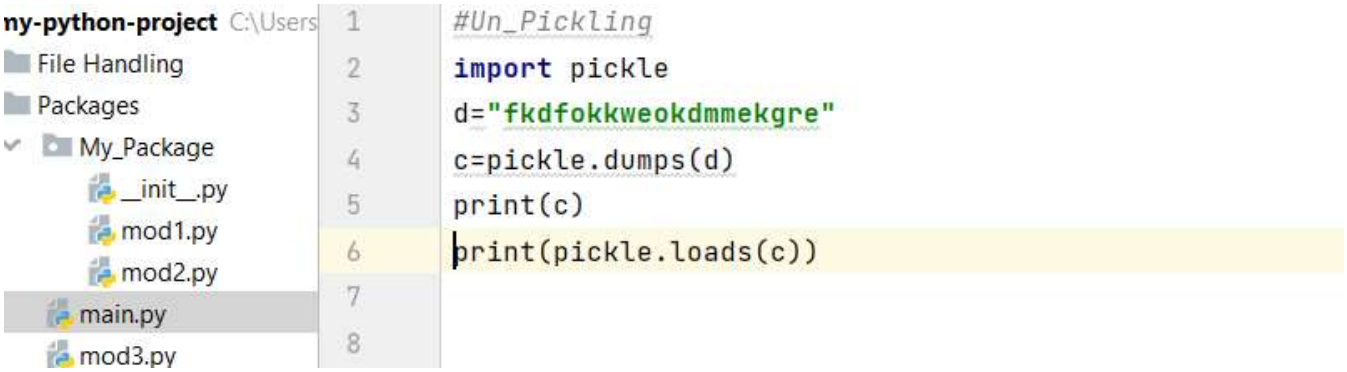
Unpickling-

- The process that takes a binary array and converts it to an object hierarchy is called unpickling.
- The unpickling process is done by using the load() function of the pickle module and returns a complete object hierarchy from a simple bytes array.
- **load ( )-**Deserializes from an open-like object.

main.py mod3.py

```python
#Un_Pickling
import pickle
with open('mod3.py','rb')as fp:
    data=pickle.load(fp)
print(data)
```

- **loads ( )**-Deserializes from a string.



- [ ] **JSON-**

- JSON(JavaScript Object Notation) has been *part of the Python standard library* is a lightweight data-interchange format.
- It is easy for humans to read and write. It is easy to parse and generate.
- Because of its simplicity, JSON is a way by which we store and exchange data, which is accomplished through its JSON syntax, and is used in many web applications.
- As it is in human readable format, and this may be one of the reasons for using it in data transmission, in addition to its effectiveness when working with APIs.
- An example of JSON-formatted data is as follow –

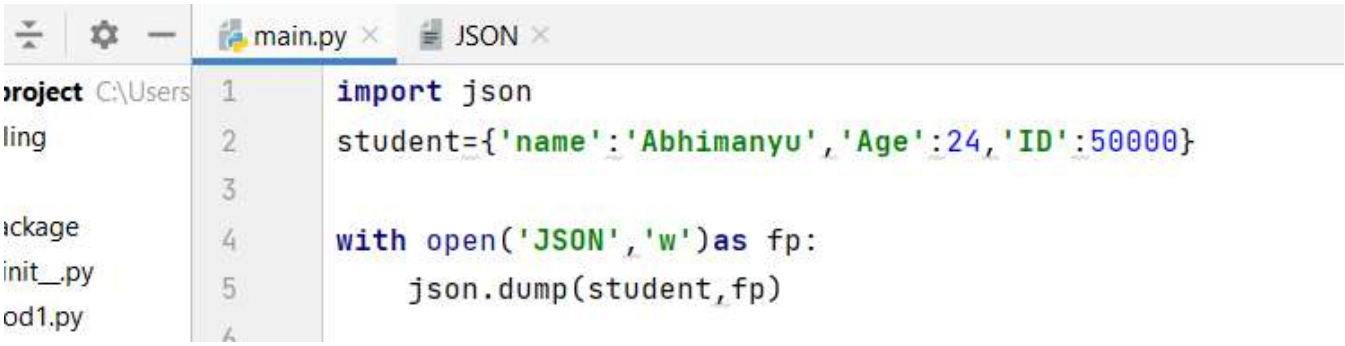
```
{"EmployID": 40203, "Name": "Zack", "Age":54, "isEmployed": True}
```

- Notation difference between Python and json

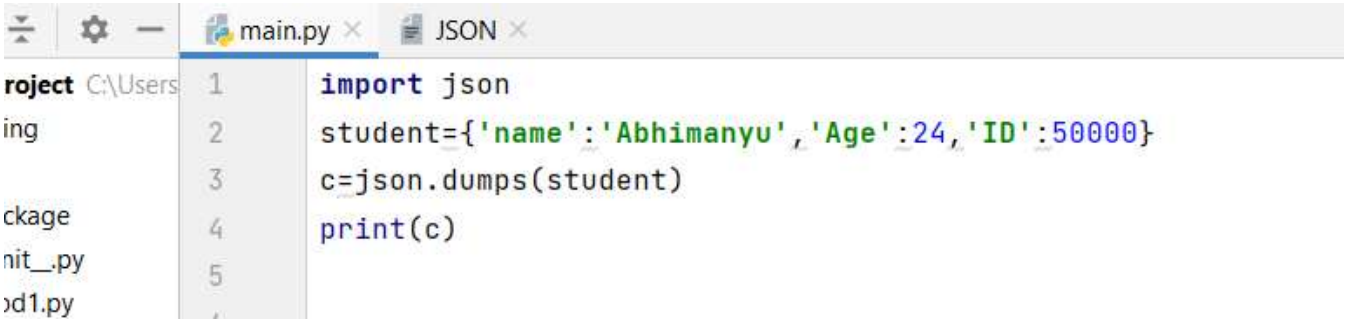
Python	JSON
Integer	Number
Float	Number
Dictionary	JSON
String	String
True	true
False	false

- **dump( )**-

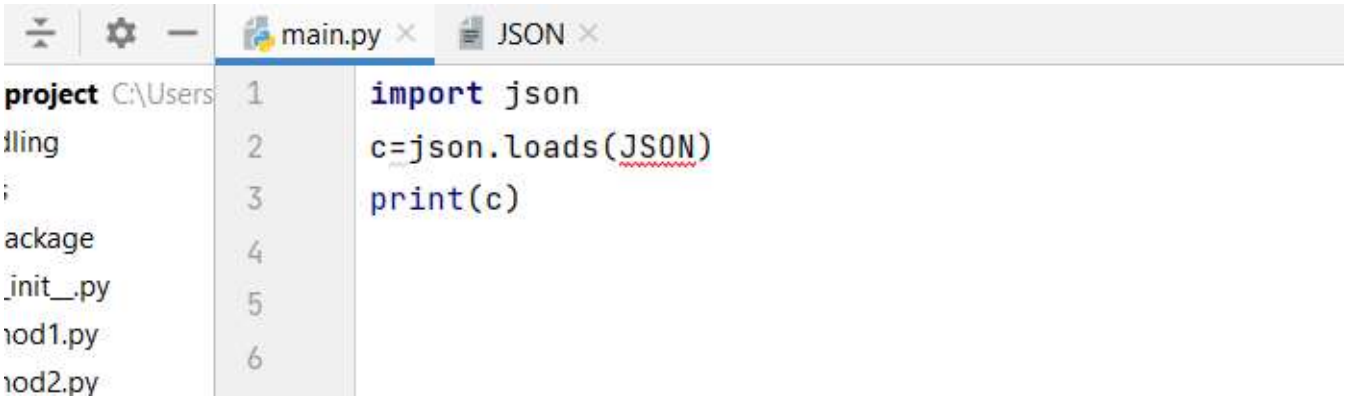
- Will convert python Dictionary to JSON string and also dumps in file.



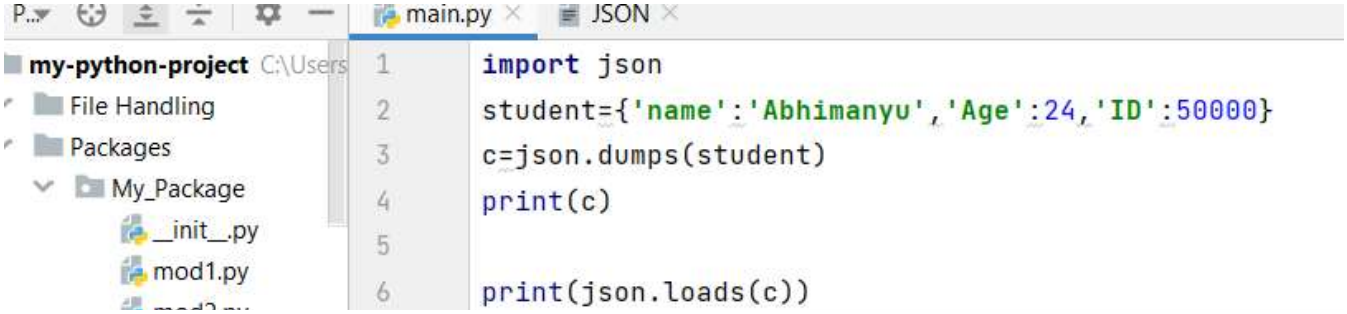
- **dumps( )-**
- Will convert python dictionary into JSON string.

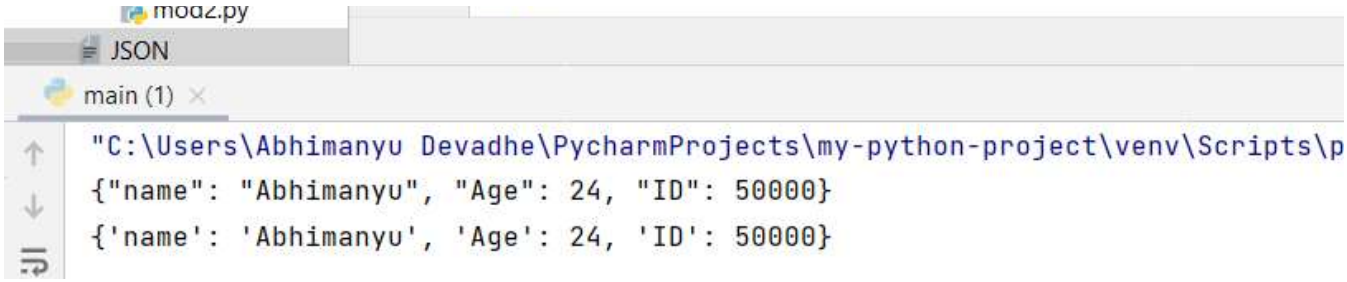


- **load( )-**
- JSON string will convert in python Dictionary.



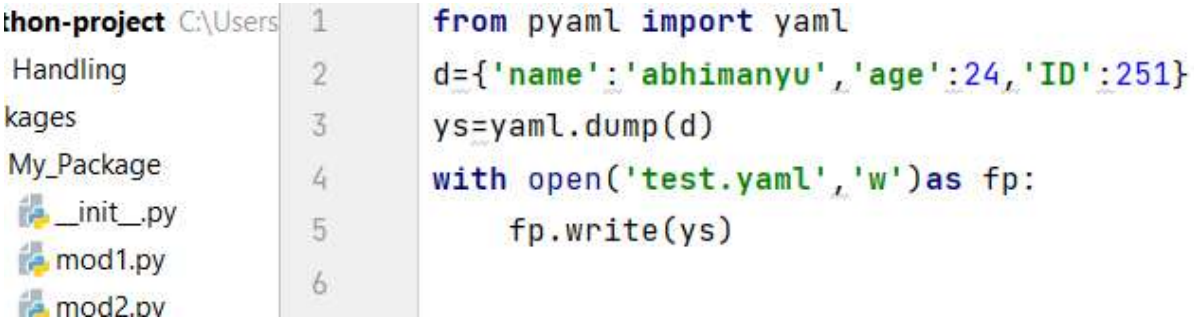
- **loads( )=**
- JSON string will convert in python Dictionary.



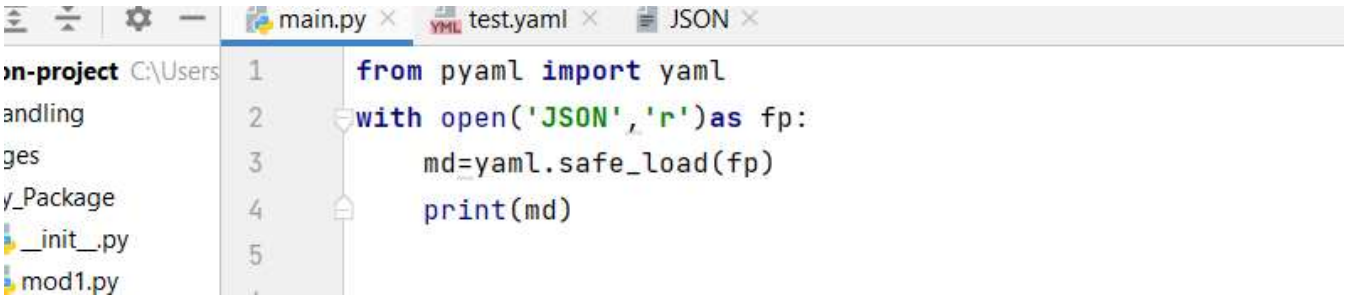


- [ ] **YAML-**
  - YAML may be the most human friendly data serialization standard for all programming languages.
  - Python yaml module is called pyaml.
  - YAML is an alternative to JSON-
  1. **Human readable code –** YAML is the most human readable format so much so that even its front-page content is displayed in YAML to make this point.
  2. **Compact code –** In YAML we use whitespace indentation to denote structure not brackets.
  3. **Syntax for relational data –** For internal references we use anchors (&) and aliases (*).
  4. **One of the area where it is used widely is for viewing/editing of data structures –** for example configuration files, dumping during debugging and document headers.

- **Serialization using YAMl**
- Syntax-yaml.dump( )objecgt



- [ ] **Deserialization Using YAML**
  - Syntax-yaml.safe_load( )
  - will convert JSON string into python dictionary.



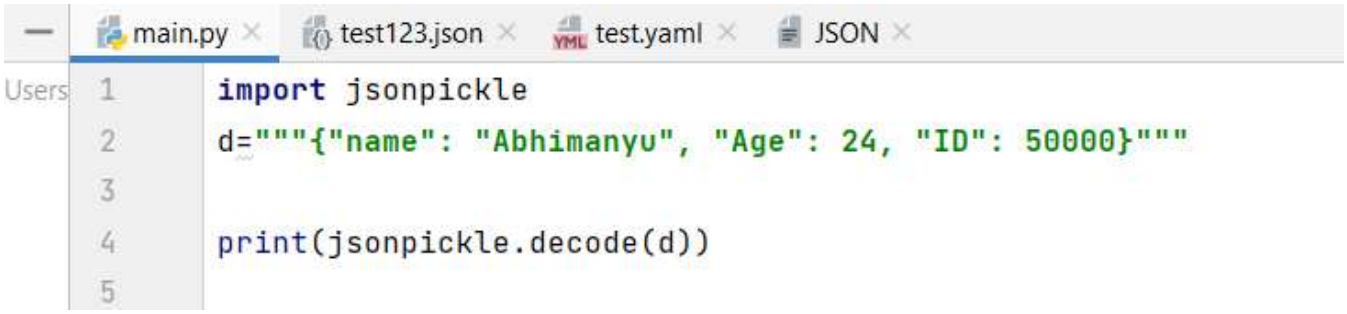
  - also we can convert YAML file into python dictobjedionary.

- [ ] **Object serialization using JSONPICKLE**

- [ ] **Object serialization using JSONPICKLE-**

- It will convert object into JSON string.
- There two methods in jsonpickle-
- **encode-**object to json string.
- syntax-jsonpickle.encode( )

- **decode-**json string will convert into object.
- Syntax-jsonpickle.decode( )



#Exceptional Handling-

- Error in Python can be of two types i.e. Syntax error and Exception Error. Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.
- ***Syntax and Exception Error.***

- [ ] Syntax Error-

- As the name suggests this error is caused by the wrong syntax in the code. It leads to the termination of the program.

- [ ] Exception Error-

- Exceptions are raised when the program is syntactically correct, but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

- [ ] **Try and Except Statement – Catching Exceptions :-**

- Try and except statements are used to catch and handle exceptions in Python.
- Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

```python
a=[1,2,3]

try:
    print("The third element from Array is= ",(a[2]))
    print("The fourth element from Array is= ", (a[3]))
except:
    print("element is not availble in Array")
```

Output-

```
"C:\Users\Abhimanyu Devadhe\PycharmProjects\my-python-project\ve
The third element from Array is=  3
element is not availble in Array
```

- In the above example, the statements that can cause the error are placed inside the try statement (second print statement in our case). The second print statement tries to access the fourth element of the list which is not there and this throws an exception. This exception is then caught by the except statement.

- [ ] **Catching Specific Exception-**

- A try statement can have more than one except clause, to specify handlers for different exceptions. *Note that at most one handler will be executed.*
- The general syntax for adding specific exceptions are –

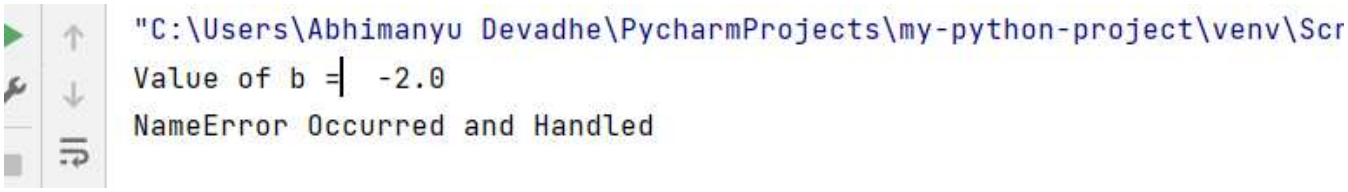
```
try:
    # statement(s)
except IndexError:
    # statement(s)
except ValueError:
    # statement(s)
```

Example- Program to handle multiple errors with one except statement.

```python
# Program to handle multiple errors with one
# except statement
# Python 3
def fun(a):
    if a < 4:
        # throws ZeroDivisionError for a = 3
        b = a / (a - 3)
    # throws NameError if a >= 4
    print("Value of b = ", b)
try:
    fun(2)
    fun(5)
```

```
# note that braces () are necessary here for
# multiple exceptions
except ZeroDivisionError:
    print("ZeroDivisionError Occurred and Handled")
except NameError:
    print("NameError Occurred and Handled")
```

Output-

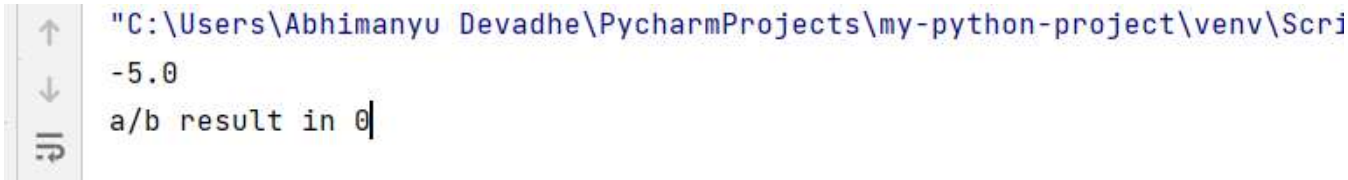


- [ ] **Try with Else Clause-**

- In python, you can also use the else clause on the try-except block which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.
- Example-

```
# Program to depict else clause with try-except
# Python 3
# Function which returns a/b
def AbyB(a, b):
    try:
        c = ((a + b) / (a - b))
    except ZeroDivisionError:
        print("a/b result in 0")
    else:
        print(c)
# Driver program to test above function
AbyB(2.0, 3.0)
AbyB(3.0, 3.0)
```

Output-



- [ ] **Finally keyword in python-**
  - Python provides a keyword finally, which is always executed after the try and except blocks. The final block always executes after normal termination of try block or after try block terminates due to some exception.
  - Syntax-

```
try:
    # Risky Code.....

except:
    # optional block..
    # Handling of exception (if required)

else:
    # execute if try don't have exception

finally:
    # Some code .....(always executed)
    #Clean-up purpose
```

Example-

```
# Python program to demonstrate finally
# No exception Exception raised in try block
try:
    k = 5 // 0  # raises divide by zero exception.
    print(k)
# handles zerodivision exception
except ZeroDivisionError:
    print("Can't divide by zero")
finally:
    # this block is always executed
    # regardless of exception generation.
    print('This is always executed')
```

Output-

```
"C:\Users\Abhimanyu Devadhe\PycharmProjects\my-python-project\ve
can't be divided by 0
This is always be executed
```

- [ ] **Raising Exception-**
  - The Raise statement allows the programmer to force a specific exception to occur. The sole

argument in raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).

- Example-

```
# Program to depict Raising Exception
try:
    raise NameError("Hi there")  # Raise Error
except NameError:
    print("An exception")
    raise  # To determine whether the exception was raised or not
```

- Output-

```
"C:\Users\Abhimanyu Devadhe\PycharmProjects\my-python-project\venv\Scripts\python.exe" '
An exception
Traceback (most recent call last):
  File "C:\Users\Abhimanyu Devadhe\PycharmProjects\my-python-project\Packages\main.py",
    raise NameError("Hi there")  # Raise Error
NameError: Hi there
```

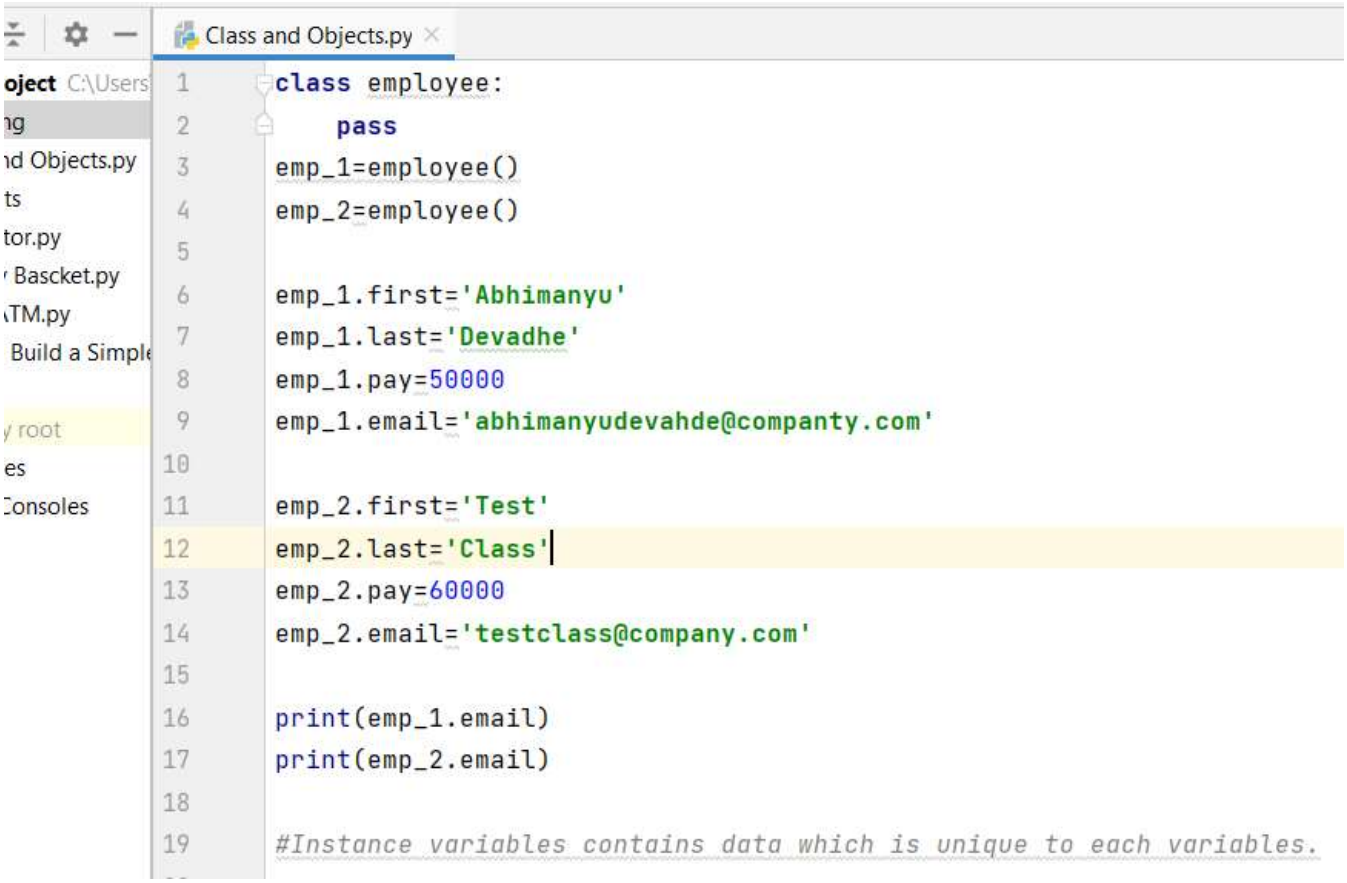
Classes and Objects-

- **Classes-**
- Classes is basically used to group our data and functions logically in a way that's easy to reuse and also easy to build upon if needed.
- 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.
- To understand the need for creating a class let's consider an example, let's say you wanted to track the number of dogs that may have different attributes like breed, age. If a list is used, the first element could be the dog's breed while the second element could represent its age. Let's suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it's the exact need for classes.
- Class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.
- **Some points on Python class:**
  - Classes are created by keyword class.
  - Attributes are the variables that belong to a class.
  - Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

```
Class Definition Syntax:
```

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

- ***Example-Simple class and creating instances.***



- [ ] **Class Objects-**

- An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with *actual values*.
- An object consists of :
  - **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
  - **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
  - **Identity:** It gives a unique name to an object and enables one object to interact with other objects.
- Declaring Objects-When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

```
class Dog:
```

```
class Dog:
    #A simple class
    #attribute
    attr1 = "mammal"
    attr2 = "dog"

    #A sample method
    def fun(self):
        print("I'm a", self.attr1)
        print("I'm a", self.attr2)

#Driver code
#Object instantiation
Rodger = Dog()

#Accessing class attributes
#and method through objects
print(Rodger.attr1)
Rodger.fun()
```

- [ ] ***The 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.
- Self is nothing but object reference.
- If we have a method that takes no arguments, then we still have to have one argument.
- This is similar to this pointer in C++ and this reference in Java.
- When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

- [ ] ***__init__ Method-***

- The __init__ method is similar to constructors in C++ and Java. Constructors are used to initializing the object's state. Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation.
- It runs as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.
- Example-Initialize class attributes and create methods by using __init__ method.

```
class Student:
    # self is object reference
    # first,last,email,pay-is variables from class
    def __init__(self,first,last,email,id):
        self.first = first
        self.last=last
        self.email=email
        self.id=id

#stud_1 and stud_2 are the variables in class
stud_1=Student('Abhimanyu','Devadhe','abhimanyudevadhe@company.com',525)
stud_2=Student('Sudarshan','Devadhe','Sudarshandevadhe@company.com',625)
```

```
print(stud_1.email)
print(stud_2.id)
```

- Output-

- [ ] **Class/Static and Instance Variables in Python-**

- Used declare variables within a class. There are two main types: class variables, which have the same value across all class instances (i.e. static variables), and instance variables, which have different values for each object instance.

```
class Car(object):
    wheels = 4  #Class/Static Variable,value shared across all class instance.

#Instance variables,value specific to instance. Instance variables are usually initialized in
methods.     def __init__(self, make):
        self.make = make

newCar = Car("Honda")

#accessing instance variable
print ("My new car is a {}".format(newCar.make))

#accessing class variable
print ("My car, like all cars, has {} wheels".format(Car.wheels))
```

- [ ] **Class Methods and Static Method-**

- **Regular Methods**-Automatically passed the instance as the first argument and we call that ***'self'***.
- **Class Methods**-Automatically passed the class as the first argument and we call that ***'cls'***.
- Example-

```
class employee:
    raise_amount=1.04
    num_of_emps=0

    def __init__(self,first,last,pay):
        self.first=first
        self.last=last
        self.pay=pay
        self.email=first+'.'+last+'company.com'
        employee.num_of_emps+=1

    def fullname(self):
        return '{} {}'.format(self.first,self.last)
```

```
    def apply_raise(self):
        self.pay=int(self.pay*employee.raise_amount)
    @classmethod
    def set_raise_amt(cls,amount):
        cls.raise_amount=amount

emp_1=employee('Abhimanyu','Devadhe',50000)
emp_2=employee('Test','Class',60000)

emp_1.set_raise_amt(1.06)
print(employee.raise_amount)
print(emp_1.raise_amount)
print(emp_2.raise_amount)
```

- We can use class methods as ***"Alternative Constructor".***
- Example-Parse Employee information which is in string and separated by hyphens.

```
    raise_amount=1.04
    num_of_emps=0

    def __init__(self,first,last,pay):
        self.first=first
        self.last=last
        self.pay=pay
        self.email=first+'.'+last+'company.com'
        employee.num_of_emps+=1

    def fullname(self):
        return '{} {}'.format(self.first,self.last)
    def apply_raise(self):
        self.pay=int(self.pay*employee.raise_amount)
    @classmethod
    def set_raise_amt(cls,amount):
        cls.raise_amount=amount
    @classmethod #Alternative Constructor
    def from_string(cls,emp_str):
        first, last, pay = emp_str.split('-')
        return cls(first, last, pay)

emp_1=employee('Abhimanyu','Devadhe',50000)
emp_2=employee('Test','Class',60000)

emp_str_1='Jon-doe-70000'
emp_str_2='Abhi-dev-55000'
emp_str_3='Akash-dev-45000'

new_emp_1=employee.from_string(emp_str_1)
print(new_emp_1.fullname())

emp_1.set_raise_amt(1.06)
```

- **Static Methods**- Static method don't pass anything automatically.
- Example-is workday of not.

```
class employee:
    raise_amount=1.04
    num_of_emps=0
```

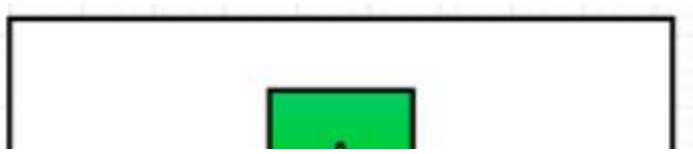
```
    def __init__(self,first,last,pay):
        self.first=first
        self.last=last
        self.pay=pay
        self.email=first+'.'+last+'company.com'
        employee.num_of_emps+=1

    def fullname(self):
        return '{} {}'.format(self.first,self.last)
    def apply_raise(self):
        self.pay=int(self.pay*employee.raise_amount)
    @classmethod
    def set_raise_amt(cls,amount):
        cls.raise_amount=amount
    @staticmethod
    def is_workday(day):
        if day.weekday()== 5 or day.weekday()==6:
            return False
        else:
            return True


emp_1=employee('Abhimanyu','Devadhe',50000)
emp_2=employee('Test','Class',60000)
import datetime

my_date=datetime.date(2016,10,8)
print(employee.is_workday(my_date))
```

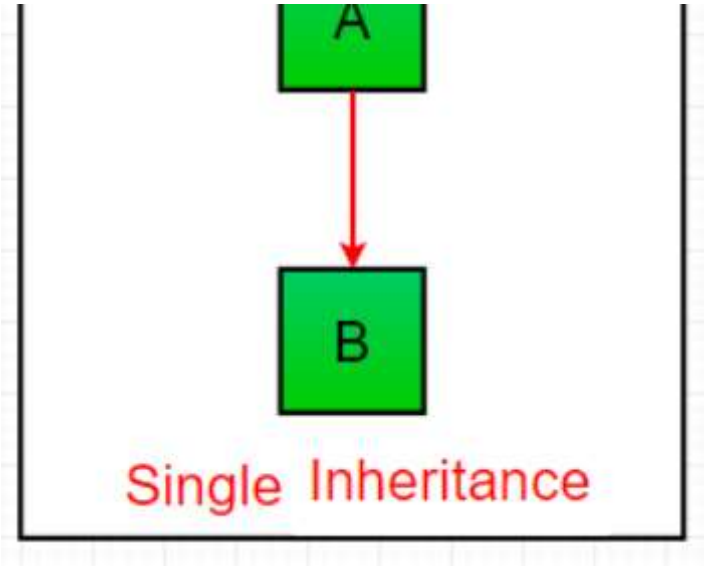
- [ ] **Class Inheritance-**

- Inheritance allows us to inherit attributes and methods from a parent class.With the help of inheritance we can create subclasses and get all the functionality of our parent class and we can write or overwrite new functionality without affecting parent class.
- Inheritance is defined as the capability of one class to derive or inherit the properties from some other class and use it whenever needed.
- Inheritance provides the following properties:
    - It represents real-world relationships well.
    - It provides reusability of 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.

- [ ] **Types Of Inheritance-**

- **Single Inheritance-**Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.

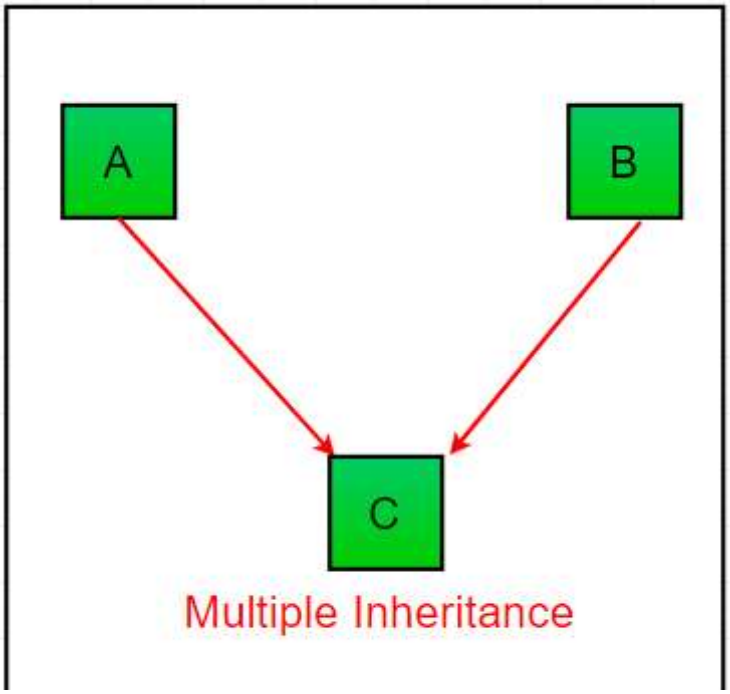




- Example-

```
1  class parant:
2      def fun(self):
3          print('This is from parant class')
4
5  class child(parant):
6      def fun2(self):
7          print('This is from child class')
8
9  obj=child()
10 obj.fun2()
11 obj.fun()
```

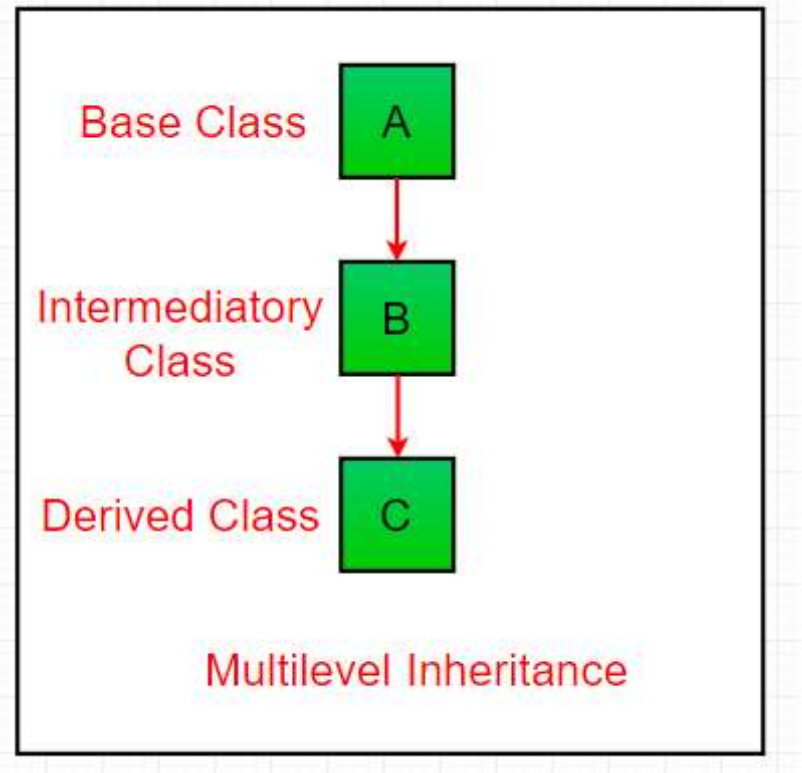
- **Multiple Inheritance-**When a class can be derived from more than one base class this type of inheritance is called multiple inheritance. In multiple inheritance, all the features of the base classes are inherited into the derived class.



- Example-

```
class Father:
    fathername=" "
    def father(self):
        print('Father name is',self.fathername)
class Mother:
    mothername=""
    def mother(self):
        print('Mother name is ',self.mothername)

class Son(Father,Mother):
    def parants(self):
        print('Father name is ',self.fathername)
        print('Mother name is ',self.mothername)


s1=Son()
s1.fathername='Ram'
s1.mothername='Seeta'
s1.parants()
```

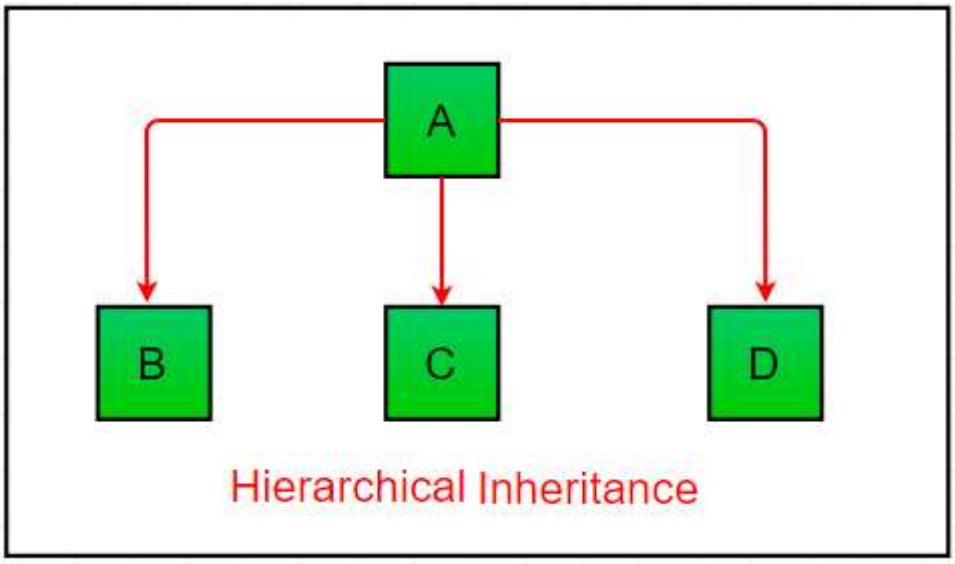
- **Multilevel Inheritance** -In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and grandfather.



- Example-

```
class Grandfather:
    def __init__(self,grandfathername):
        self.grandfathername=grandfathername
class Father(Grandfather):
    def __init__(self,fathername,grandfathername):
        self.fathername=fathername
        Grandfather.__init__(self,grandfathername)
class Son(Father,Grandfather):
    def __init__(self,sonname,fathername,grandfathername):
        self.sonname=sonname
        Father.__init__(self,fathername,grandfathername)

s1=Son('Abhimanyu','Vasantrao','kakasaheb')

print('Son name is',s1.sonname)
print('Father name is ',s1.fathername)
print('Grandfather name is ',s1.grandfathername)
print(s1.grandfathername)
print(s1.fathername)
```

- **Hierarchical Inheritance-**When more than one derived classes are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.



- Example-

```
class Parant:
    def fun1(self):
        print('This is from Parant Class')
class Son_1(Parant):
    def fun2(self):
```

```
        print('This is from Son_1')
class Son_2(Parant):
    def fun3(self):
        print('This is from Son_2')

obj1=Son_1()
obj2=Son_2()

obj1.fun1()
obj1.fun2()
print(Son_1.mro())
obj2.fun1()
obj2.fun3()
Son_2.mro()
print(Son_2.mro())
```

- **Hybrid Inheritance-** Inheritance consisting of multiple types of inheritance is called hybrid inheritance.
- Example-

```
class School:
    def fun1(self):
        print('This is from fun1')
class Stud_1(School):
    def fun2(self):
        print('This is from fun2')
class Stud_2(School):
    def fun3(self):
        print('This is from fun3')

class Stud_3(Stud_2,Stud_1):
    def fun4(self):
        print('This is from fun4')

obj=Stud_3()

obj.fun1()
obj.fun2()
obj.fun3()
obj.fun4()
```

#Super ( ) Method-

- One of the important OOP features is Inheritance in python. When a class inherits some or all of the behaviors from another class is known as Inheritance. In such a case, the inherited class is the subclass and the latter class is the parent class.
- *In an inherited subclass, a parent class can be referred to with the use of the super() function. The super function returns a temporary object of the superclass that allows access to all of its methods to its child class.*
- The benefits of using a super function are:-

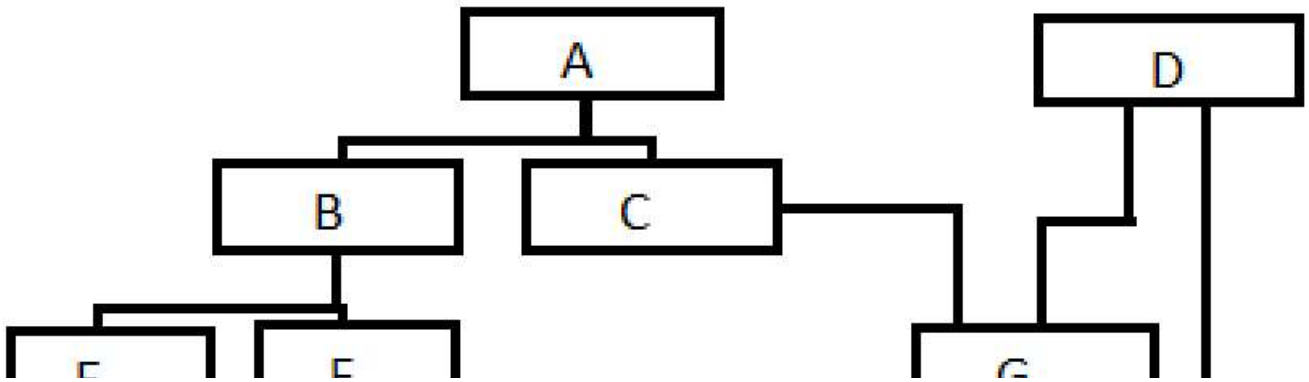
1. Need not remember or specify the parent class name to access its methods. This function can be used both in single and multiple inheritances.
2. This implements modularity (isolating changes) and code reusability as there is no need to rewrite the entire function.
3. Super function in Python is called dynamically because Python is a dynamic language unlike other languages.

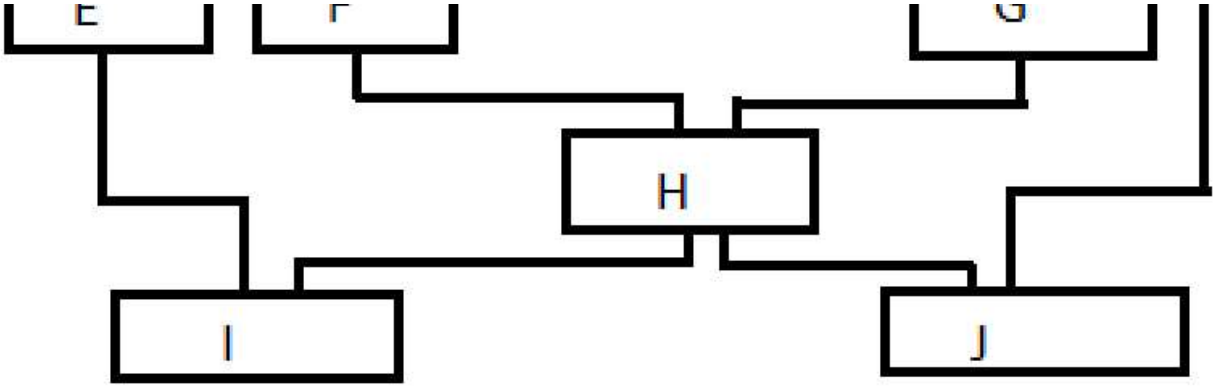
- Example-

```
class Test:
    def m1(self):
        print('m1 from Test')
class Test1(Test):
    def m1(self):                # m1 method from Test is over write in Test1
        super().m1()             #If we have to use m1 from Test,we can use super( ).
        print('m1 from Test1')

obj=Test1()
obj.m1()
```

Method Resolution Order-

- As we know that, a class that being inherited is called the Subclass or Parent class, while the class that inherits is known as a child class or subclass. In the multi-inheritance, a class can consist of many functions, so the method resolution order technique is used to search the order in which the base class is executed.
- In simple words - "The method or attributes is explored in the current class, if the method is not present in the current class, the search moves to the parent classes, and so on". This is an example of a depth-first search.
- MRO- ***self-->Left parent-->Right Parent-->Root Parent-->Class object-->Error***





- Find MRO of I,J,H,E,F,G-

1. MRO of I= I-->E -->H -->F -->B-->G-->C-->A-->D-->Class Objects
2. MRO of J= j-->--H-->F-->B --> G-->C -->A -->D -->Class Objects
3. MRO of H=H--> F-->B -->G -->C -->A -->D -->Class Objects
4. MRO of E=E-->B-->A--Class Objects
5. MRO of F=F-->B-->A-->Class Objects
6. MRO of G=G-->C-->A-->D-->Class Objects

- Find MRO using Pycharm-

```
class A:
    def m1(self):
        print('From A')
class B(A):
    def m1(self):print('From B')
class C(A):
    def m1(self):print('From C')
class D:
    def m1(self):print('From D')
class E(B):
    def m1(self):print('From E')
class F(B):
    def m1(self):print('From F')
class G(C,D):
    def m1(self):print('From G')
class H(F,G):
    def m1(self):print('From H')
class I(E,H):
    def m1(self):print('From I')
class J(H,D):
    def m1(self):print('From J')
obj=I()
obj.m1()
print('MRO of I: ',I.mro())
print('MRO of J: ',J.mro())
print('MRO of H: ',H.mro())
print('MRO of E: ',E.mro())
print('MRO of F: ',F.mro())
print('MRO of G: ',G.mro())
```

```
From I
MRO of I: [<class '__main__.I'>, <class '__main__.E'>, <class '__main__.H'>, <class '__main__.F'>, <class '__main__.B'>, <clas
```

```
MRO of I:  [<class '__main__.I'>, <class '__main__.E'>, <class '__main__.H'>, <class '__main__.F'>, <class '__main__.B'>, <clas
s '__main__.G'>, <class '__main__.C'>, <class '__main__.A'>, <class '__main__.D'>, <class 'object'>]
MRO of J:  [<class '__main__.J'>, <class '__main__.H'>, <class '__main__.F'>, <class '__main__.B'>, <class '__main__.G'>, <clas
s '__main__.C'>, <class '__main__.A'>, <class '__main__.D'>, <class 'object'>]
MRO of H:  [<class '__main__.H'>, <class '__main__.F'>, <class '__main__.B'>, <class '__main__.G'>, <class '__main__.C'>, <clas
s '__main__.A'>, <class '__main__.D'>, <class 'object'>]
MRO of E:  [<class '__main__.E'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
MRO of F:  [<class '__main__.F'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
MRO of G:  [<class '__main__.G'>, <class '__main__.C'>, <class '__main__.A'>, <class '__main__.D'>, <class 'object'>]
```

Abstraction -

- *Abstraction in python* is defined as a *process of handling complexity by hiding unnecessary information from the user.*
- This is one of the core concepts of object-oriented programming (OOP) languages. That enables the user to implement even more complex logic on top of the provided abstraction without understanding or even thinking about all the hidden background/back-end complexity.
- For development perspective Abstraction used for security purpose.means user can't be call method with accidently creating object.
- In python we can implement Abstraction using Abstract base class.
- The method which is defined in Abstract class is called as Abstract method.
- To make abstract method we have to need Decorator.
- ***Using Abstract class when we have to make list of Abstract methods in specific class,we can only create objects of these methods until all methods to be implement.***
- Abstract class only have list of methods or definitions .

```
from abc import abstractmethod,ABC

class Demo(ABC):       #Abstract class only have list of methods or definitions.
    @abstractmethod
    def fun(self):
        pass
    @abstractmethod
    def fun1(self):
        pass
class Demo1(Demo):      #Implementation of Methods
    def fun(self):
        print("In fun")
    def fun1(self):
        print('In fun1')

obj=Demo1()
obj.fun1()
obj.fun()
```

Access Specifier-

- Various object-oriented languages like C++, Java, Python control access modifications which are *used to restrict access to the variables and methods of the class*. Most programming languages has three forms of access modifiers, which are **Public**, **Protected** and **Private** in a class.

- Python uses '_' symbol to determine the access control for a specific data member or a member function of a class. Access specifiers in Python have an important role to play in securing data from unauthorized access and in preventing it from being exploited.
- A Class in Python has three types of access modifiers:

- [ ] **Public Access Modifier-**

- The members of a class that are declared public are easily accessible from any part of the program. All data members and member functions of a class are public by default.
- *We can access Variables and methods from class is publically or anywhwere in program .*

- [ ] **Protected Access Modifier-**

- The members of a class that are declared protected are only accessible to a class derived from it. Data members of a class are declared protected by adding a single underscore '_' symbol before the data member of that class.
- *We can access Variables and methods from class is anywhere in class i.e inside the class,outside the class and inside the child class.*

- [ ] **Private Access Modifier-**

- The members of a class that are declared private are accessible within the class only, private access modifier is the most secure access modifier. Data members of a class are declared private by adding a double underscore '__' symbol before the data member of that class.
- *We can access Variables and methods from class is only within the class.*
- With the help of '*name mangling' concept we can access variable or method from Private Access Modifier outside the class.x*
- Syntax of Name Mangling-
  (object._class name__Private variable/Method name)
- Example for Access specifier for Variables-

```
class Demo():
    a=100
    _b=200
    __c=300
    def sample(self):  #We can access all three variables within the class.
        print(Demo.a)
        print(Demo._b)
        print(Demo.__c)

obj=Demo()
print("Inside The class")
obj.sample()

print("Outside The class") #We access only a and b variables outside the class.
print(obj.a)
print(obj._b)
print(obj._Demo__c)  #For access c variable outside the class,we have to use name mangling.
```

- Example for Access specifier for Methods-

```
class Operations:
    x=""
    y=""
    def add(self,x,y):
        self.x=x
        self.y=y
```

```
        print("Addition of x and y is:",x+y)
    def _sub(self,x,y):
        self.x = x
        self.y = y
        print("Substraction of x and y is:", x - y)
    def __mul(self,x,y):
        self.x = x
        self.y = y
        print("Multiplication of x and y is:", x * y)

obj=Operations()

obj.add(15,15)
obj._sub(20,15)
obj._Operations__mul(15,5) #For access __mul method outside the class,we have to use name mangling.
```

- [ ] **Polymorphism-**
  - The word polymorphism means having many forms. In programming, *polymorphism means the same function name (but different signatures) being used for different types.*
  - Polymorphism in python is used for a common function name that can be used for different types. This concept is widely applied in object-oriented based python programming. Like other programming languages say Java, C+, *polymorphism is also implemented in python for different purpose commonly Duck Typing, Operator overloading and Method overloading, and Method overriding.*
  - This polymorphism process can be achieved in two main ways namely overloading and overriding.
- [ ] **Overloading-**
  - In overloading we add functionality in method,constructor or in Operator which have same name but have different arguments.
  - Basically there are 3 types of Overloading 1) Operator 2)Method 3)Constructor Overloading.

#Operator Overloading-

- Operator overloading is the type of overloading in which an operator can be used in multiple ways beyond its predefined meaning.
- Eaxmaple-

```
class Sample:
    def __init__(self,a):
        self.a=a
    def __add__(self, other): #__add__ is a magic method which is inbuilt method.
        return self.a+other.a

obj=Sample(10)
obj1=Sample(20)
print(obj+obj1)
```

#Method Overloading-

- Python does not support function overloading. When we define multiple functions with the same name, the later one always overrides the prior and thus, in the namespace, there will always be a single entry against each function name.
- In java method overloading is possible due to java is not Dynamically typed,although python is dynamically typed.
- There is no possible magic method,so we cant overload Method.
- Example-

```
class Sample:
    def m1(self):
        print("In M1")
    def m1(self):                #Created method with same name.
        print('from second in M1')

obj=Sample()

obj.m1()
```

- Output-

```
"C:\Users\Abhimanyu Devadhe\PycharmProjects\my-python-project\venv\Scripts\pyth
from second in M1
```

#Constructor Overloading-

- Constructor overloading means more than one *constructor* in a class with the same name but a different argument (parameter). Python does not support Constructor overloading; it has no form of function.
- In Python, Methods are defined solely by their name, and there can be only one method per class with a given name.
- There is no possible magic method,so we cant overload constructor.
- Example-

```
class Test:
    def __init__(self):
        print("No argument")
    def __init__(self,a):       #Constructor is override
        print("One argument")
    def __init__(self,a,b):     #Constructor is overrided,we have to pass two arguments a and b.
        print("Two arguments")

obj=Test(10,20)
```

- [ ] ## Overriding-

- Overriding is *the property of a class to change the implementation of a method provided by one of its base classes.*

- Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. *In other language we can say that in overriding we can implement method provided by base class and also add more functions in method or we can update base class.*

#Method Overriding-

- Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.
- When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to **override** the method in the super-class.
- Example-

```
class T:
    def m1(self):
        print("From version 1")
class T1(T):
    def m1(self):
        super().m1()
        print("From Version 2")
class T2(T1):
    def m1(self):
        super().m1()
        print("From version 3")

obj=T2()
obj.m1()
```

#Constructor Overriding-

- Sometimes you want to override the inherited __init__ function.
- When you override the constructor, the constructor from the parent class, from which we inherited, is not called at all. If you still want that functionality, you have to call it yourself. This is done with super(): it returns a reference to the parent class, so we can call the constructor of the parent class.

```
class Test:
    def __init__(self):
        print("No argument")
    def __init__(self):      #Constructor is override
        print("One argument")
    def __init__(self):     #Constructor is overrided.
        print("Two arguments")

obj=Test()
```

#Difference between Method Overloading and Overriding Method.

S.NO	Method Overloading	Method Overriding
1.	In the method overloading, methods or functions must have the same name and different signatures.	Whereas in the method overriding, methods or functions must have the same name and same signatures.
2.	Method overloading is a example of compile time polymorphism.	Whereas method overriding is a example of run time polymorphism.
3.	In the method overloading, inheritance may or may not be required.	Whereas in method overriding, inheritance always required.
4.	Method overloading is performed between methods within the class.	Whereas method overriding is done between parent class and child class methods.
5.	It is used in order to add more to the behavior of methods.	Whereas it is used in order to change the behavior of exist methods.
6.	In method overloading, there is no need of more than one class.	Whereas in method overriding, there is need of at least of two classes.

#Multi-Threading-

- Multithreading is a way of achieving multitasking. In multithreading, the concept of **threads** is used.
- In computing, a process is an instance of a computer program that is being executed. Any process has 3 basic components:
  - An executable program.
  - The associated data needed by the program (variables, work space, buffers, etc.)
  - The execution context of the program (State of process)
- *A thread is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS (Operating System).*
- In simple words, a thread is a sequence of such instructions within a program that can be executed independently of other code. For simplicity, you can assume that a thread is simply a subset/subpart of a process. Process is made of threads.
- **Multithreading** is defined as the ability of a processor to execute multiple threads concurrently.
- **Multithreahding** is possible only when your code is independent or your threads/functions from code is independent.
- For get of current thread-

```
import threading

print(threading.currentThread().getName())
```

- There are three ways for using Multithreading-

1. Creating thread without using any class or Function base threading.

- Example 1

```
import threading
def sample():
    print("From sample function")
def sample1():
    print("From sample1 function")

t1=threading.Thread(target=sample)    #Creating Threads t1 and t2
t2=threading.Thread(target=sample1)

t1.start()
t2.start()

print('From main thread')
```

- **Join( ) Method in Multithreading**

1. *Join method in Python allows Main thread to wait until another sub threads completes its execution. In simpler words, it means it waits for the Sub threads to die.*
2. The Thread. Join() method is used to call a thread and blocks the calling thread until a thread terminates i.e Join method waits for finishing other threads by calling its join method.
3. Join ( ) method is used to wait the main thread untill the child threads complete its execution.
4. Example-

```
import time
from threading import Thread
def squr(n):
    for i in n:
        time.sleep(1)
        print('Squr of:',i,'is',i*i)
def cube(n):
    for i in n:
        time.sleep(1)
        print('Cube of:',i,'is',i*i*i)

n=list(range(10))
starttime=time.time()
t1=Thread(target=squr,args=(n,))
t2=Thread(target=cube,args=(n,))
t1.start()
t2.start()
t1.join()
t2.join()
# squr(n)
# cube(n)
endtime=time.time()
```

```
print('Total Time:',endtime-starttime)
```

2.Creating Threads by Extending Thread class

- **Run ( ) Method-**run() method is an inbuilt method of the Thread class of the threading module in Python. *This method is used to represent a thread's activity. It calls the method expressed as the target argument in the Thread object along with the positional and keyword arguments taken from the args and kwargs arguments, respectively.*
- Example-

```
from threading import Thread
class Test(Thread):
    def run(self):
        print("From Child class")

t1=Test()
t1.start()
print('From Main Thread')
```

3.Creating threads without extending Thread class-

- Example-

```
from threading import Thread
class Test:
    def show(self):
        for i in range(10):
            print("From child 1")
    def m1(self):
        for i in range(10):
            print("From child 2")
obj=Test()
t1=Thread(target=obj.show)
t2=Thread(target=obj.m1)
t1.start()
t2.start()
t1.join()
t2.join()

print('All threds are are proceeded')
```

Logging In Python-

- Logging is a python module in the standard library that provides the facility to work with the framework for releasing log messages from the Python programs.
- Logging is used to tracking events that occur when the software runs.
- This module is widely used by the developers when they work to logging. It is very important tool which used in software development, running, and debugging.

- Logging is beneficial to store the logging records. Suppose there is no logging record, and the program is interrupted during its execution, we will be unable to find the actual cause of the problem.
- Somehow, we detect the cause of the crash but it will consume a lot of time to resolve this. Using the logging, we can leave a trace of breadcrumbs so that if the problem happens in the program, we can find the cause of the problem easily.
- We can face many problems while running applications such as we suppose an integer, and we have been given a float, the service is under maintenance and many more. These problems are hard to determine and time-consuming.

 # How Logging Works..??

- The logging is a powerful module used by the beginners as well as enterprises. This module provides a proficiency to organize different control handlers and a transfer log messages to these handlers.

To releasing a log message, we need to import the logging module as follows.

import logging

- Now, we will call the logger to log messages that we want to see. The logging module offers the five levels that specify the severity of events. Each event contains the parallel methods that can be used to log events at the level of severity. Let's understand the following events and their working.

1. **DEBUG -** It is used to provide detailed information and only use it when there is diagnosing problems.
2. **INFO -** It provides the information regarding that things are working as we want.
3. **WARNING -** It is used to warn that something happened unexpectedly, or we will face the problem in the upcoming time.
4. **ERROR -** It is used to inform when we are in some serious trouble, the software hasn't executed some programs.
5. **CRITICAL -** It specifies the serious error, the program itself may be incapable of remaining executing.

- The above levels are sufficient to handle any types of problems. These corresponding numerical values of the levels are given below.

Level	Numeric Values
NOTSET	0
DEBUG	10
INFO	20
WARNING	30
ERROR	40
CRITICAL	50

Let's have a look at the several logger objects offered by the module itself

- **Logger.info(msg) :** It is used to log a message with level INFO on this logger.
- **Logger.warning(msg) :** It is used to log a message with level WARNING on this logger.

- **Logger.warning(msg) :** It is used to log a message with level WARNING on this logger.
- **Logger.error(msg) :** It is used to log a message with level ERROR on this logger.
- **Logger.critical(msg) :** It is used to log a message with level CRITICAL on this logger.
- **Logger.log(lvl,msg) :** It is used to logs a message with integer level lvl on this logger.
- **Logger.exception(msg) :** It is used to log a message with level ERROR on this logger.
- **Logger.setLevel(lvl) :** It is used to sets the beginning of this logger to lvl. It will ignore all the messages which are written below.
- **Logger.addFilter(filt) :** It is used to add a specific filter filt to the to this logger.
- **Logger.removeFilter(filt) :** It is used to eliminates a specific filter filt to the to this logger.
- **Logger.filter(record) :** It put on the filter of logger to the record. If the record available and to be handled then returns True. Otherwise, it will return False.
- **Logger.addHandler(hdlr) :** It is used to add a particular handler hdlr to the to this logger.
- **Logger.removeHandler(hdlr) :** It is used to eliminate a particular handler hdlr to this logger.
- **Logger.hasHandlers() :** It is used to verify if the logger contains any handler configured or not.

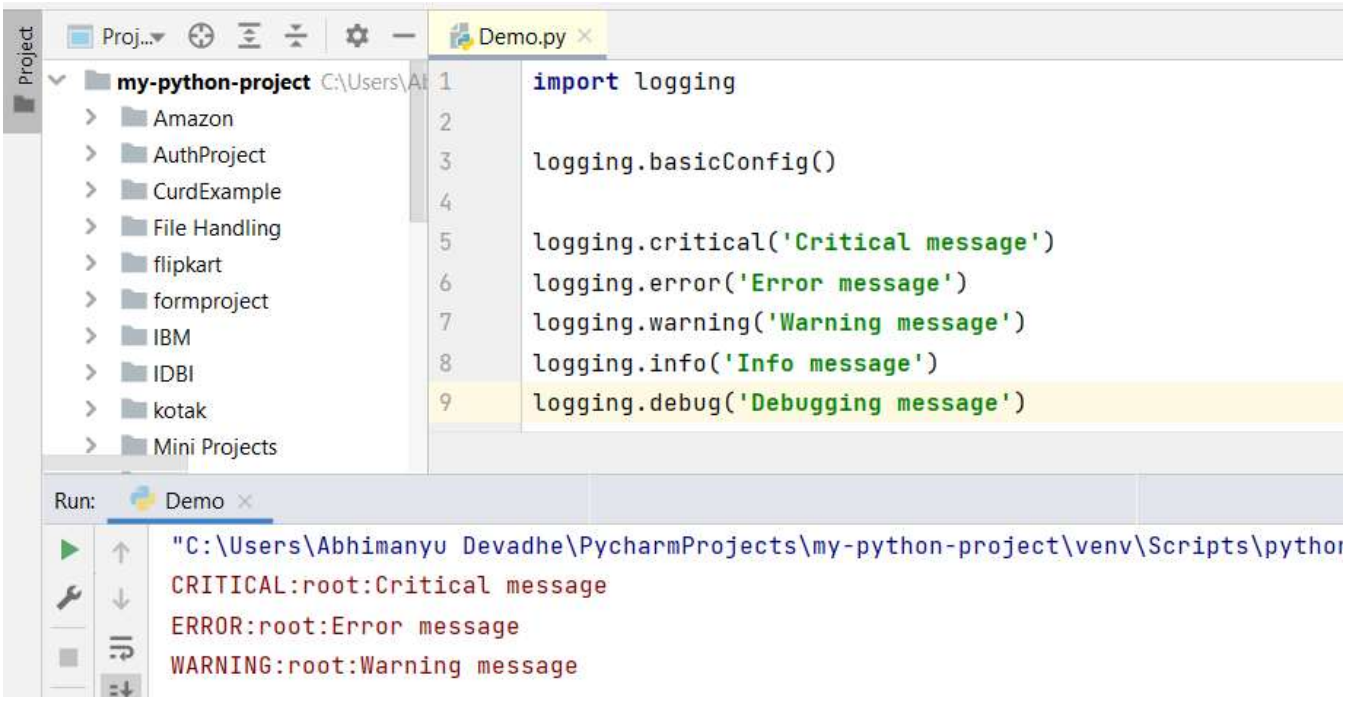
- [ ] **Basic Configurations-**

- The main task of logging is to store the records events in a file. The logging module provides the **basicConfig(**kwarg)**, used to configure the logging.
- It accepts some of the commonly used argument as follows.

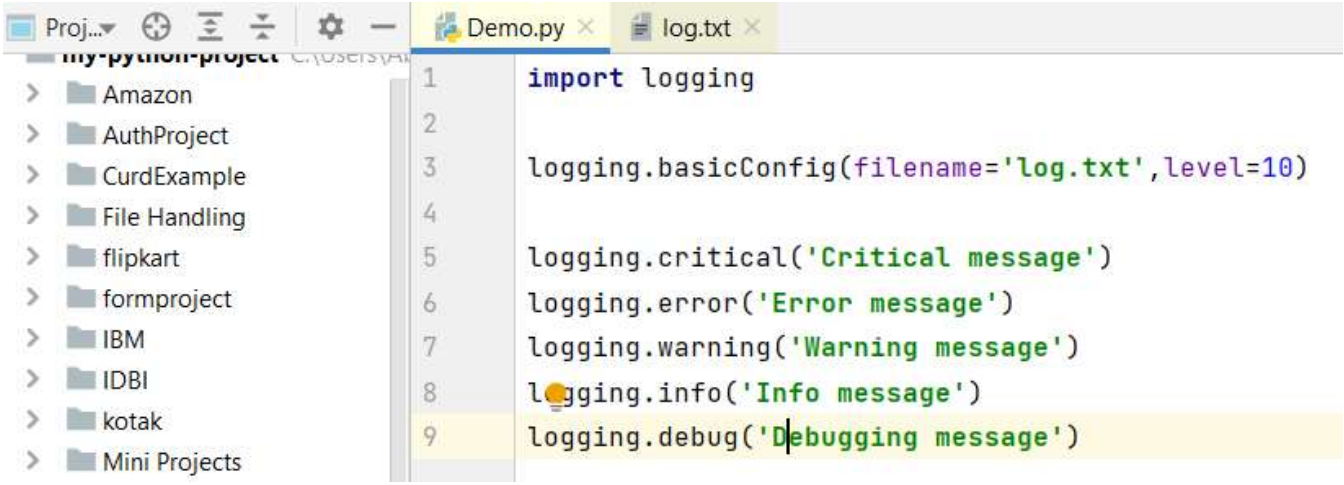
1. **level -** The specified severity level is set by the root level.
2. **filename -** It specifies a file.
3. **filemode -** It opens a file in a specific mode. The default mode of the opening file is a, which means we can append the content.
4. **format -** The format defines the format of the log message.

- We can set the level of log messages by using the level parameter as we want to record. We need to pass the one constant in the class, which would permit all logging calls.

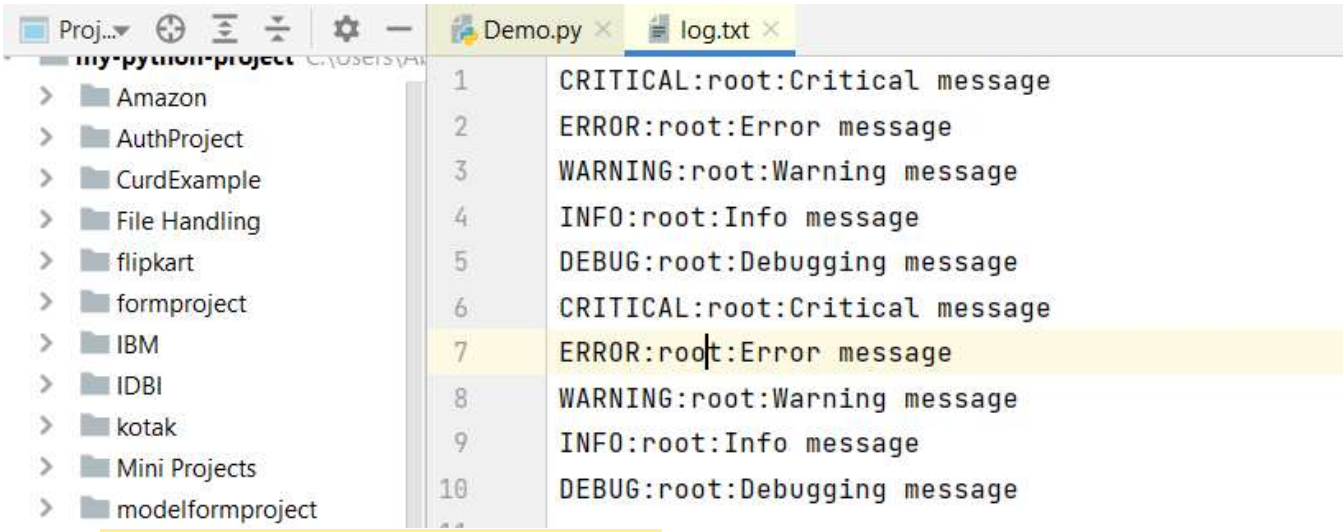
- When we create basic configuration,from above 5 level we only get warning and above level.



- Create log.txt and configure filemode-



- Created log.txt

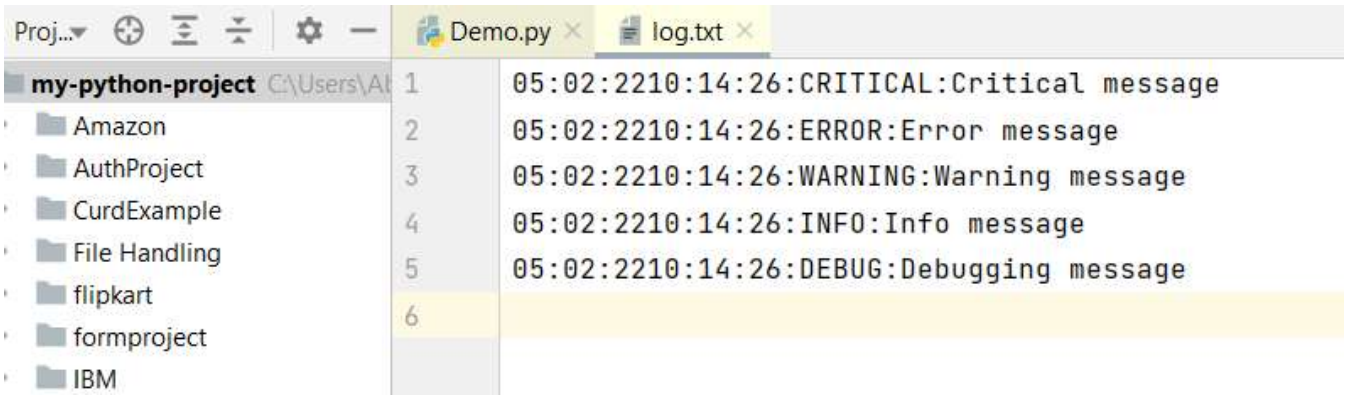


- **Note- By default file is on append mode.**
- Add filemode and desired location -

- Formatting Date time and Logging messages-

```
import logging

logging.basicConfig(filename='log.txt',level=10,filemode='w',
                    format='%(asctime)s:%(levelname)s:%(message)s',
                    datefmt='%d:%m:%y%I:%M:%S')
```

```
logging.critical('Critical message')
logging.error('Error message')
logging.warning('Warning message')
logging.info('Info message')
logging.debug('Debugging message')
```



- [ ] **Classes and Functions in Logging-**

- We have seen so far the default logger called **root**. The logging module is used it whenever its functions are called such as **logging.debug(),** logging.error(), etc. We can also define own logger by creating an object of the **Logger** class. Here, we are defining the commonly used classes and functions.
  Below are the classes and functions defined in the logging module.
  - **Logger -** The logger object is used to call the functions directly.
  - **LogRecord -** It creates automatically log record file which consists the information related to all event of being logged such as the logger's name, the function, the line number, the message, and more.
  - **Handler -** The handlers are used to dispatch the **LogRecord** to the output endpoint. The **FileHandler, StreamHandler, HTTPHandler, SMTTPHandler** are the subclasses of a **Handler**.
  - **Formatters -** The formatters are used to define the structure of the output. It is used the string formatting methods to specify the format of the log messages.
- If we don't have the message to format, the default is to use the raw message. The default format date format is.
  %Y-%m-%d %H:%M:%S
- The following format is used to make the log message in the human -readable format.
  '%(asctime)s - %(levelname)s - %(message)s'

- If We have two Packages/Modules/files and we have to maintain log in one file.
- In log file we will get Packages/Modules/files names along with Error -
  logger=logging.getLogger(os.path.basename(__file__))

```python
import os
logging.basicConfig(filename='log.txt',level=10)
logger=logging.getLogger(os.path.basename(__file__))
logger.critical('Critical message')
logger.error('Error message')
try:
    a
except Exception as e:
    logger.exception('Please Check Variable')
logger.warning('Warning message')
logger.info('Info message')
logger.debug('Debugging message')
```

Demo.py | Demo 1.py | log.txt

```python
import logging
import os
logging.basicConfig(filename='log.txt',level=10)
logger=logging.getLogger(os.path.basename(__file__))
logger.critical('Critical message')
logger.error('Error message')
try:
    a=10
    a.endswith('fjeiefi')
except Exception as e:
    logger.exception('please check variable')

logger.warning('Warning message')
logger.info('Info message')
logger.debug('Debugging message')
```

Demo.py | Demo 1.py | log.txt

```
CRITICAL:Demo.py:Critical message
ERROR:Demo.py:Error message
ERROR:Demo.py:Please Check Variable
Traceback (most recent call last):
  File "C:\Users\Abhimanyu Devadhe\PycharmProjects\my-python-project\venv\Mudules\Demo.py", li
    a
NameError: name 'a' is not defined
WARNING:Demo.py:Warning message
INFO:Demo.py:Info message
DEBUG:Demo.py:Debugging message
CRITICAL:Demo 1.py:Critical message
ERROR:Demo 1.py:Error message
ERROR:Demo 1.py:please check variable
Traceback (most recent call last):
  File "C:\Users\Abhimanyu Devadhe\PycharmProjects\my-python-project\venv\Mudules\Demo 1.py",
    a.endswith('fjeiefi')
AttributeError: 'int' object has no attribute 'endswith'
```

```
AttributeError: 'int' object has no attribute 'endswith'
WARNING:Demo 1.py:Warning message
INFO:Demo 1.py:Info message
DEBUG:Demo 1.py:Debugging message
```

- [ ] **Work with Handlers-**

- Handlers are generally used to configure logger and transmit the logs to the many places at a time. It sends the log messages to the standard output stream or a file over HTTP or on email.

```
import logging

# Create a custom logger_obj
logger_obj = logging.getLogger(__name__)

# Create handlers
w_handler = logging.StreamHandler()
e_handler = logging.FileHandler('file.log')
w_handler.setLevel(logging.WARNING)
e_handler.setLevel(logging.ERROR)

# Create formatters and add it to handlers
c_format = logging.Formatter('%(name)s - %(levelname)s - %(message)s')
f_format = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
w_handler.setFormatter(c_format)
e_handler.setFormatter(f_format)

# Add handlers to the logger_obj
logger_obj.addHandler(w_handler)
logger_obj.addHandler(e_handler)

logger_obj.warning('This is a warning message')
logger_obj.error('This is an error message')
```

- Explanation-In the following program, we have created a custom logger named the **logger_obj** and created a LogRecord that stores the all record of the logging events and passed it to all the Handlers that it has: **w_handlers** and **e_handlers**.
- The w_handlers is a **stream handler** with the level WARNING. It accepts the log from the **LogRecord** to generate the output in the format string and print it to the screen.
- The **e_handler is a** file handler with the level **ERROR**. It disregards the LogRecord as its level WARNING.
- **Conclusion**-The logging module is flexible and easy to use. It is very useful for keeping track of the logging records and displaying the appropriate message to the user. It provides the flexibility to create custom log levels, handler classes, and many other useful methods.

- [ ] **Assertion In Python-**

- Python assert keyword is defined as a debugging tool that tests a condition. The Assertions are mainly the assumption that asserts or state a fact confidently in the program. For example, while writing a division function, the divisor should not be zero, and you assert that the divisor is not equal to zero.

- It is merely a Boolean expression that has a condition or expression checks if the condition returns true or false. If it is true, the program does not do anything, and it moves to the next line of code. But if it is false, it raises an **AssertionError** exception with an optional error message.
- The main task of assertions is to inform the developers about unrecoverable errors in the program like "file not found", and it is *right to say that assertions are internal self-checks for the program.* It is the most essential for the testing or quality assurance in any application development area. The syntax of the assert keyword is given below.

  assert condition, error_message(optional)

#Why Assertion is used..?

- It is a debugging tool, and its primary task is to check the condition. If it finds that the condition is true, it moves to the next line of code, and If not, then stops all its operations and throws an error. It points out the error in the code.
- **Where Assertion in Python used..??**
  - Checking the outputs of the functions.
  - Used for testing the code.
  - In checking the values of arguments.Checking the valid input.
- There are two types of Assertion-

1.Simple Assert-

- Example-

```
def sqr(n):
    return n**n

a=sqr(4)
print(a)

assert sqr(4)==16
```

2.Agumented Assert(Very simple assert)-

- In agumented Assert we can give default error message.
- Example-

```
def sqr(n):
    return n**n

a=sqr(4)
print(a)

assert sqr(4)==16,'sqr of 4 should be 16'
```

- [ ] **Interview Notes-**