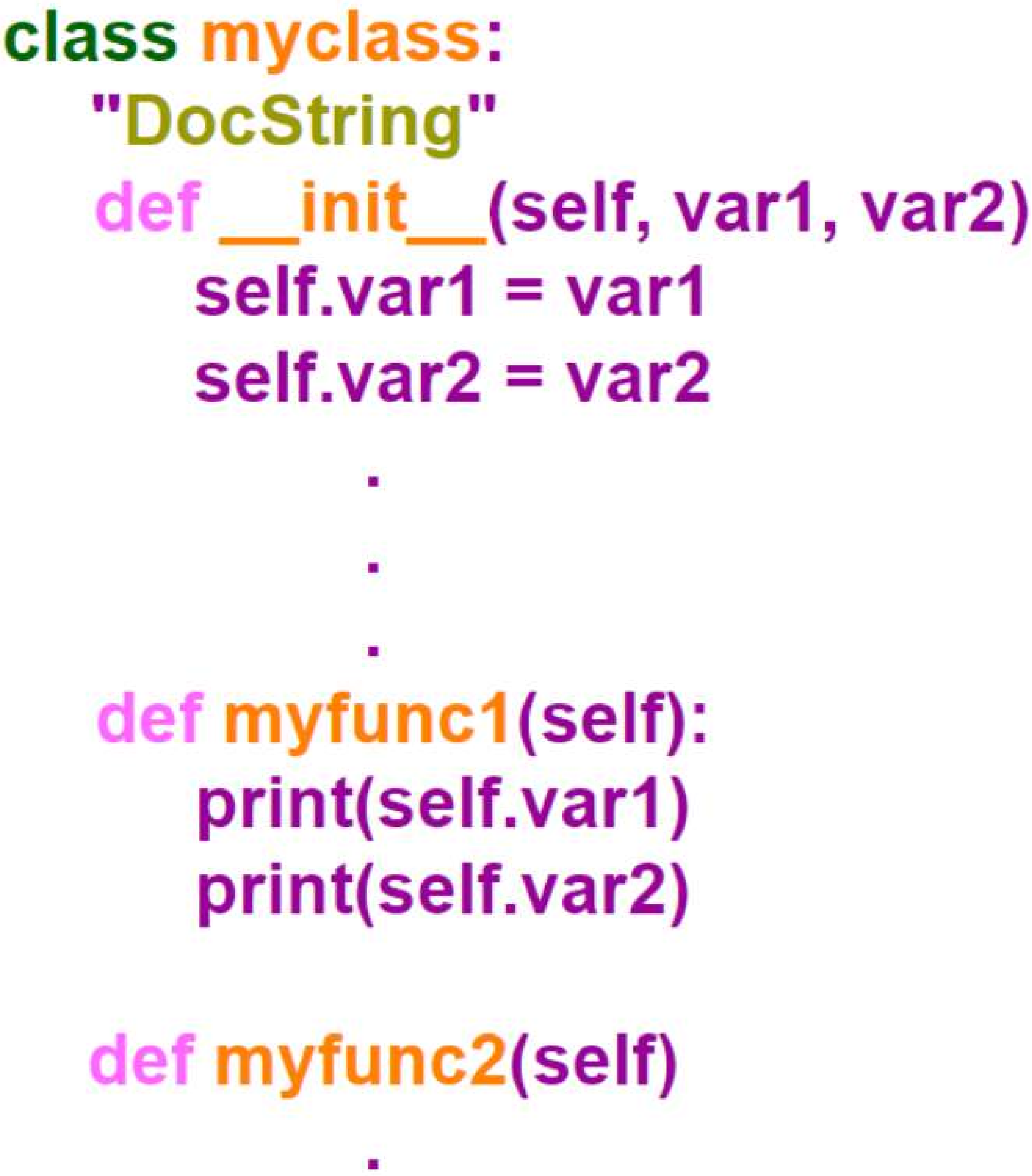
Classes & Objects

- A Class is an object constructor or a "blueprint" for creating objects.
- Objects are nothing but an encapsulation of variables and functions into a single entity.
- Objects get their variables and functions from classes.
- To create a class we use the keyword class.
- The first string inside the class is called docstring which gives the brief description about the class.
- All classes have a function called **int**() which is always executed when the class is being initiated. We can use **int**() function to assign values to object properties or other operations that are necessary to perform when the object is being created
- The self parameter is a reference to the current instance of the class and is used to access class variables.
- self must be the first parameter of any function in the class The super() builtin function returns a temporary object of the superclass that allows us to access methods of the base class.
- super() allows us to avoid using the base class name explicitly and to enable multiple inheritance.

Syntax



```
class myclass:
    "DocString"
    def __init__(self, var1, var2)
        self.var1 = var1
        self.var2 = var2
            .
            .
            .
    def myfunc1(self):
        print(self.var1)
        print(self.var2)

    def myfunc2(self)
            .
```

In [41]:

```
# Create a class with property "var1"
class myclass:
    var1 = 10

obj1 = myclass() # Create an object of class "myclass()"
print(obj1.var1)
```

```
10
```

In [42]:

```
# Create an employee class
class Employee:
    def __init__(self, name, empid): # __init__() function is used to assign v
        self.name = name
        self.empid = empid
    def greet(self): # Class Method
        print("Thanks for joining ABC Company {}!!".format(self.name))
emp1 = Employee("Raj", 34163) # Create an employee object

print('Name :- ',emp1.name)
print('Employee ID :- ',emp1.empid)
emp1.greet()
```

```
Name :-  Raj
Employee ID :-  34163
Thanks for joining ABC Company Raj!!
```

In [43]:

```
emp1.name = 'Basit' # Modify Object Properties
emp1.name
```

Out[43]:

```
'Basit'
```

In [44]:

```
del emp1.empid # Delete Object Properties
emp1.empid
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[44], line 2
      1 del emp1.empid # Delete Object Properties
----> 2 emp1.empid

AttributeError: 'Employee' object has no attribute 'empid'
```

In [45]:

```
del emp1 # Delete the object
emp1
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[45], line 2
      1 del emp1 # Delete the object
----> 2 emp1

NameError: name 'emp1' is not defined
```

In [46]:

```
emp2 = Employee("Michael", 34162) # Create an employee object
print('Name :- ',emp2.name)
print('Employee ID :- ',emp2.empid)
emp2.greet()
```

```
Name :-  Michael
Employee ID :-  34162
Thanks for joining ABC Company Michael!!
```

In [47]:

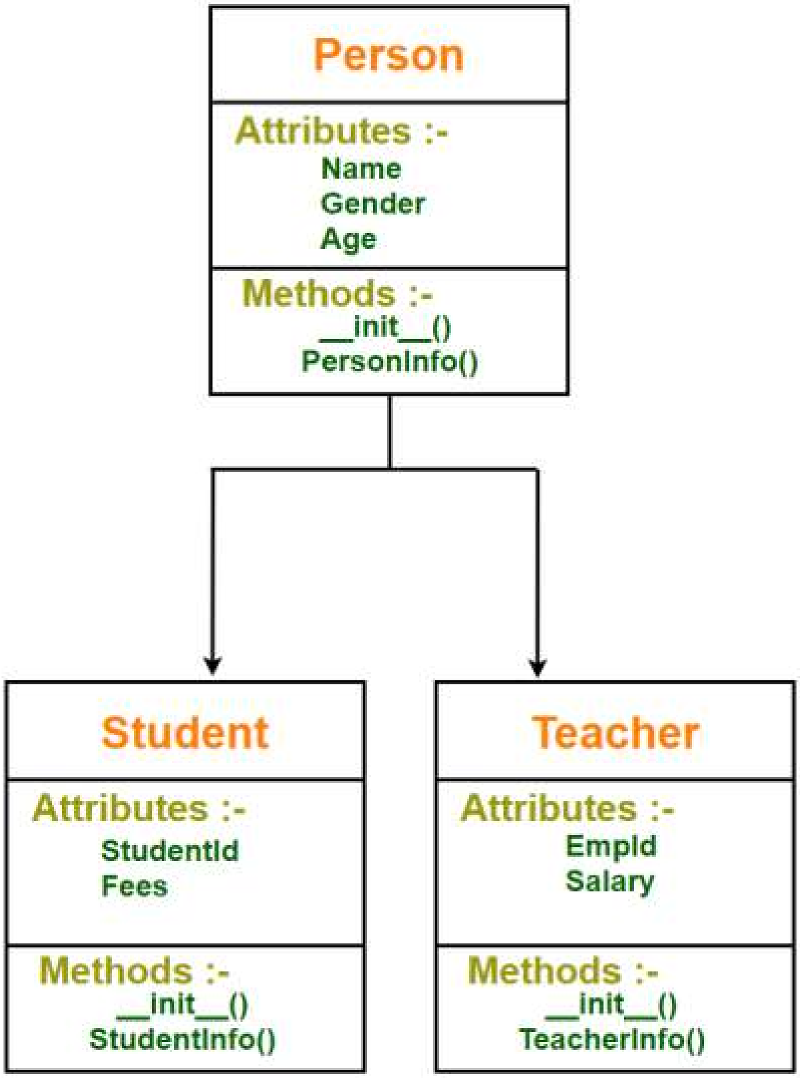
```
emp2.country = 'India' #instance variable can be created manually
emp2.country
```

Out[47]:

```
'India'
```

Inheritance

- Inheritance is a powerful feature in object oriented programming.
- Inheritance provides code reusability in the program because we can use an existing class (Super Class/ Parent Class / Base Class) to create a new class (Sub Class / Child Class / Derived Class) instead of creating it from scratch.
- The child class inherits data definitions and methods from the parent class which facilitates the reuse of features already available. The child class can add few more definitions or redefine a base class method.
- Inheritance comes into picture when a new class possesses the 'IS A' relationship with an existing class. E.g Student is a person. Hence person is the base class and student is derived class.


Person
Attributes :-
Name
Gender
Age
Methods :-
__init__()
PersonInfo()
Student
Attributes :-
StudentId
Fees
Methods :-
__init__()
StudentInfo()
Teacher
Attributes :-
EmpId
Salary
Methods :-
__init__()
TeacherInfo()

In [48]:

```
class person: # Parent Class
    def __init__(self, name , age , gender):
        self.name = name
        self.age = age
        self.gender = gender

    def PersonInfo(self):
        print('Name :- {}'.format(self.name))
        print('Age :- {}'.format(self.age))
        print('Gender :- {}'.format(self.gender))



class student(person): # Child Class
    def __init__(self,name,age,gender,studentid,fees):
        person.__init__(self,name,age,gender)
        self.studentid = studentid
        self.fees = fees
    def StudentInfo(self):
        print('Student ID :- {}'.format(self.studentid))
        print('Fees :- {}'.format(self.fees))


class teacher(person): # Child Class
    def __init__(self,name,age,gender,empid,salary):
        person.__init__(self,name,age,gender)
        self.empid = empid
        self.salary = salary

    def TeacherInfo(self):
        print('Employee ID :- {}'.format(self.empid))
        print('Salary :- {}'.format(self.salary))

stud1 = student('RAj' , 24 , 'Male' , 123 , 1200)
print('Student Details')
print('---------------')
stud1.PersonInfo() # PersonInfo() method presnt in Parent Class will be access
stud1.StudentInfo()
print()
teacher1 = teacher('Anita' , 36 , 'female' , 456 , 80000)
print('Employee Details')
print('---------------')
teacher1.PersonInfo() # PersonInfo() method presnt in Parent Class will be acc
teacher1.TeacherInfo()
```

```
Student Details
---------------
Name :- RAj
Age :- 24
Gender :- Male
Student ID :- 123
Fees :- 1200

Employee Details
---------------
Name :- Anita
Age :- 36
Gender :- female
Employee ID :- 456
Salary :- 80000
```

In [49]:

```
class person: # Parent Class
    def __init__(self, name , age , gender):
        self.name = name
        self.age = age
        self.gender = gender


    def PersonInfo(self):
        print('Name :- {}'.format(self.name))
        print('Age :- {}'.format(self.age))
        print('Gender :- {}'.format(self.gender))



class student(person): # Child Class
    def __init__(self,name,age,gender,studentid,fees):
        person.__init__(self,name,age,gender)
        self.studentid = studentid
        self.fees = fees

    def StudentInfo(self):
        print('Student ID :- {}'.format(self.studentid))
        print('Fees :- {}'.format(self.fees))


stud1 = student('RAj' , 24 , 'Male' , 123 , 1200)
print('Student Details')
print('---------------')
stud1.PersonInfo() # PersonInfo() method presnt in Parent Class will be access
stud1.StudentInfo()
print()
```

```
Student Details
---------------
Name :- RAj
Age :- 24
Gender :- Male
Student ID :- 123
Fees :- 1200
```

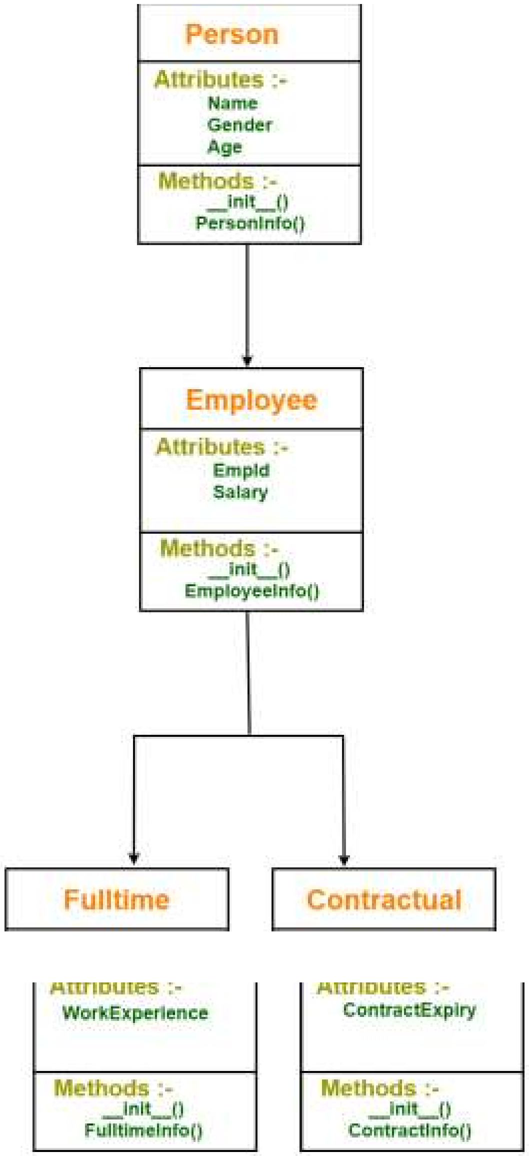
In [50]:

```
# super() builtin function allows us to access methods of the base class.
class person: # Parent Class
    def __init__(self, name , age , gender):
        self.name = name
        self.age = age
        self.gender = gender

    def PersonInfo(self):
        print('Name :- {}'.format(self.name))
        print('Age :- {}'.format(self.age))
        print('Gender :- {}'.format(self.gender))



class student(person): # Child Class
    def __init__(self,name,age,gender,studentid,fees):
        super().__init__(name,age,gender)
        self.studentid = studentid
        self.fees = fees

    def StudentInfo(self):
        super().PersonInfo()
        print('Student ID :- {}'.format(self.studentid))
        print('Fees :- {}'.format(self.fees))

stud = student('Raj' , 24 , 'Male' , 123 , 1200)
print('Student Details')
print('---------------')
stud.StudentInfo()
```

```
Student Details
---------------
Name :- Raj
Age :- 24
Gender :- Male
Student ID :- 123
Fees :- 1200
```

Multi-level Inheritance

- In this type of inheritance, a class can inherit from a child class or derived class.
- Multilevel Inheritance can be of any depth in python


Person
Attributes :-
Name
Gender
Age
Methods :-
__init__()
PersonInfo()
Employee
Attributes :-
EmpId
Salary
Methods :-
__init__()
EmployeeInfo()
Fulltime
Contractual
Attributes :-
WorkExperience
Methods :-
__init__()
FulltimeInfo()
Attributes :-
ContractExpiry
Methods :-
__init__()
ContractInfo()

In [51]:

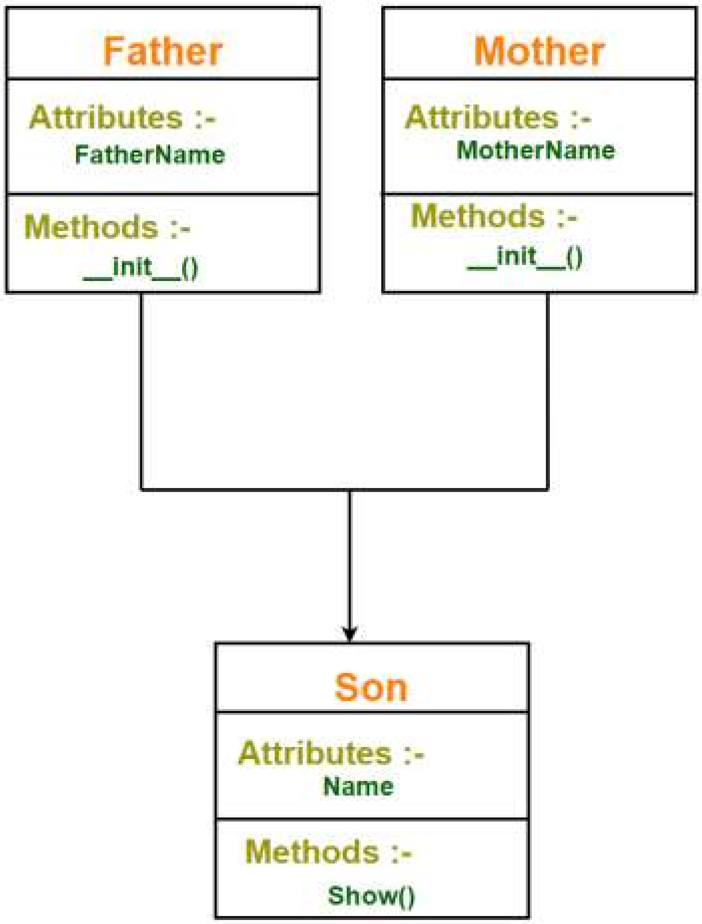
```python
class Person:
    """Parent class representing a person."""
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    def person_info(self):
        """Prints information about the person."""
        print('Name: {}'.format(self.name))
        print('Age: {}'.format(self.age))
        print('Gender: {}'.format(self.gender))


class Employee(Person):
    """Child class representing an employee."""
    def __init__(self, name, age, gender, emp_id, salary):
        super().__init__(name, age, gender)
        self.emp_id = emp_id
        self.salary = salary

    def employee_info(self):
        """Prints information about the employee."""
        print('Employee ID: {}'.format(self.emp_id))
        print('Salary: {}'.format(self.salary))


class FullTime(Employee):
    """Grandchild class representing a full-time employee."""
    def __init__(self, name, age, gender, emp_id, salary, work_experience):
        super().__init__(name, age, gender, emp_id, salary)
        self.work_experience = work_experience

    def full_time_info(self):
        """Prints information specific to a full-time employee."""
        print('Work Experience: {}'.format(self.work_experience))


class Contractual(Employee):
    """Grandchild class representing a contractual employee."""
    def __init__(self, name, age, gender, emp_id, salary, contract_expiry):
        super().__init__(name, age, gender, emp_id, salary)
        self.contract_expiry = contract_expiry

    def contract_info(self):
        """Prints information specific to a contractual employee."""
        print('Contract Expiry: {}'.format(self.contract_expiry))



# Example usage:
print('Contractual Employee Details')
print('****************************')
contract1 = Contractual('Anita', 36, 'Female', 456, 80000, '21-12-2021')
contract1.person_info()
contract1.employee_info()
contract1.contract_info()
```

```
print('\n')

print('Fulltime Employee Details')
print('****************************')
# Corrected the class name to FullTime
fulltim1 = FullTime('Raj', 22, 'Male', 567, 70000, 12)
fulltim1.person_info()
fulltim1.employee_info()
fulltim1.full_time_info()
```

```
Contractual Employee Details
****************************
Name: Anita
Age: 36
Gender: Female
Employee ID: 456
Salary: 80000
Contract Expiry: 21-12-2021


Fulltime Employee Details
****************************
Name: Raj
Age: 22
Gender: Male
Employee ID: 567
Salary: 70000
Work Experience: 12
```

Multiple Inheritance

- Multiple inheritance is a feature in which a class (derived class) can inherit attributes and methods from more than one parent class.
- The derived class inherits all the features of the base case.


Father
Attributes :-
FatherName
Methods :-
__init__()
Mother
Attributes :-
MotherName
Methods :-
__init__()
Son
Attributes :-
Name
Methods :-
Show()

In [52]:

```python
# Super Class
class Father:
    def __init__(self):
        self.fathername = str()

# Super Class
class Mother:
    def __init__(self):
        self.mothername = str()

# Sub Class
class Son(Father, Mother):
    name = str()

    def __init__(self):
        super().__init__()  # Calling the constructors of parent classes

    def show(self):
        print('My Name: ', self.name)
        print("Father: ", self.fathername)
        print("Mother: ", self.mothername)

# Creating an instance of Son
s1 = Son()
s1.name = 'Bill'
s1.fathername = "John"
s1.mothername = "Kristen"
s1.show()
```

```
My Name:  Bill
Father:  John
Mother:  Kristen
```

In [53]:

```python
class Date:
    def __init__(self, date):
        self.date = date

class Time:
    def __init__(self, time):
        self.time = time

class timestamp(Date, Time):
    def __init__(self, date, time):
        Date.__init__(self, date)
        Time.__init__(self, time)
        self.datetime = self.date + ' ' + self.time
        print(self.datetime)

# Creating an instance of timestamp
datetime1 = timestamp('2020-08-09', '23:48:55')
```

```
2020-08-09 23:48:55
```

Method Overriding

- Overriding is a very important part of object oreinted programming because it makes inheritance exploit its full power.
- Overriding is the ability of a class (Sub Class / Child Class / Derived Class) to change the implementation of a method provided by one of its parent classes.
- When a method in a subclass has the same name, same parameter and same return type as a method in its super-class, then the method in the subclass is said to override the method in the super-class.
- The version of a method that is executed will be determined by the object that is used to invoke it.
- If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.

In [54]:

```python
class Person:  # Parent Class
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    def greet(self):
        print("Hello Person")


class Student(Person):  # Child Class
    def __init__(self, name, age, gender, student_id, fees):
        super().__init__(name, age, gender)
        self.student_id = student_id
        self.fees = fees

    def greet(self):
        print("Hello Student")


# Creating an instance of Student
stud = Student('Gabriel', 56, 'Male', 45, 345678)
# Calling greet() method on stud
stud.greet()  # greet() method defined in subclass will be triggered as "stud"

# Creating an instance of Person
person1 = Person('Gabriel', 56, 'Male')
# Calling greet() method on person1
person1.greet()
```

```
Hello Student
Hello Person
```

Container

- Containers are data structures that hold data values.
- They support membership tests which means we can check whether a value exists in the container or not.
- Generally containers provide a way to access the contained objects and to iterate over them.
- Examples of containers include tuple, list, set, dict, str

In [55]:

```python
list1 = ['Raj' , 'john' , 'Michael' , 'Vaibhav']
'asif' in list1 # Membership check using 'in' operator
```

Out[55]:

```
False
```

In [56]:
```
assert 'john' in list1 # If the condition returns true the program does nothin
```

In [57]:
```
assert 'john1' in list1 # If the condition returns false, Assert will stop the
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[57], line 1
----> 1 assert 'john1' in list1

AssertionError: 
```

In [58]:
```
mydict = {'Name':'Raj' , 'ID': 12345 , 'DOB': 1991 , 'Address' : 'Hilsinki'}
mydict
```

Out[58]:
```
{'Name': 'Raj', 'ID': 12345, 'DOB': 1991, 'Address': 'Hilsinki'}
```

In [59]:
```
'Raj' in mydict # Dictionary membership will always check the keys
```

Out[59]:
```
False
```

In [60]:
```
'Name' in mydict # Dictionary membership will always check the keys
```

Out[60]:
```
True
```

In [61]:
```
'DOB' in mydict
```

Out[61]:
```
True
```

In [62]:
```
mystr = 'Vaibhav'
'bh' in mystr # Check if substring is present
```

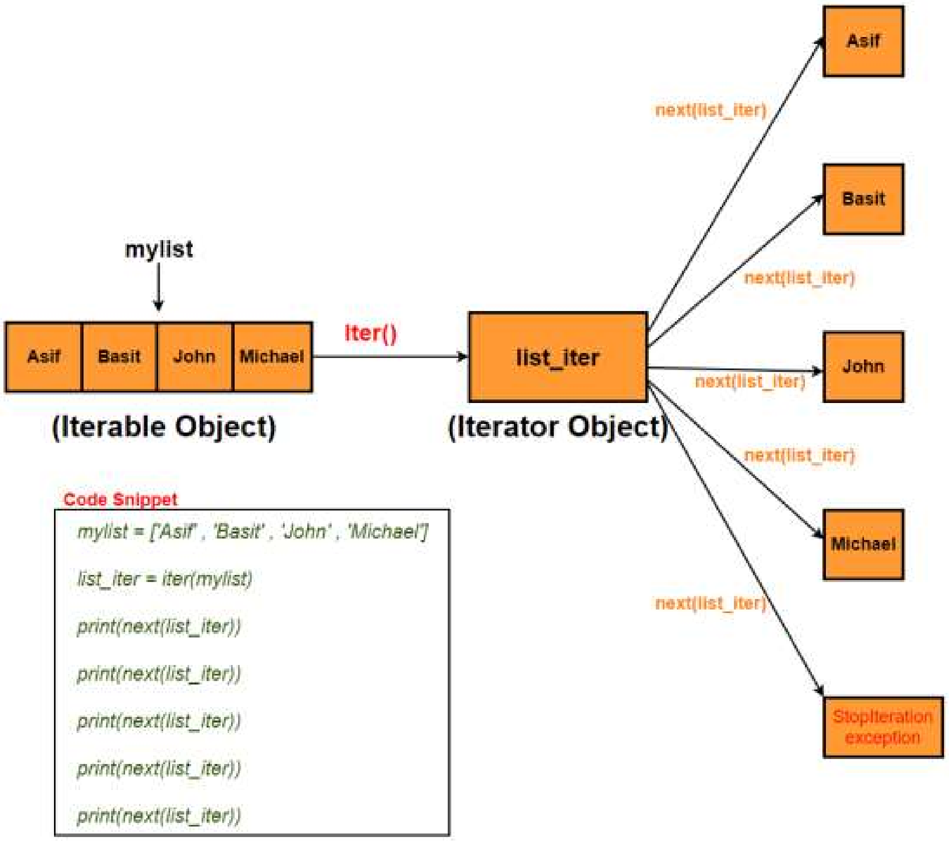
Out[62]:
```
True
```

Iterable & Iterator

In [63]:

```
- An iterable is an object that can be iterated upon. It can return an iterato
purpose of traversing through all the elements of an iterable.
- An iterable object implements __iter()__ which is expected to return an iter
iterator object uses the __next()__ method. Every time next() is called next e
iterator stream is returned. When there are no more elements available StopIte
exception is encountered. So any object that has a __next()__ method is called
- Python lists, tuples, dictionaries and sets are all examples of iterable obj
```

```
  Cell In[63], line 1
    - An iterable is an object that can be iterated upon. It can return an it
erator object with the
         ^
SyntaxError: invalid syntax
```



```
mylist = ['Asif' , 'Basit' , 'John' , 'Michael']
list_iter = iter(mylist)
print(next(list_iter))
print(next(list_iter))
print(next(list_iter))
print(next(list_iter))
print(next(list_iter))
```

In [64]:

```python
mylist = ['Raj' , 'Vaibhav' , 'John' , 'Michael']
list_iter = iter(mylist) # Create an iterator object using iter()
print(next(list_iter)) # return first element in the iterator stream
print(next(list_iter)) # return next element in the iterator stream
print(next(list_iter))
print(next(list_iter))
print(next(list_iter))
```

```
Raj
Vaibhav
John
Michael

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[64], line 7
      5 print(next(list_iter))
      6 print(next(list_iter))
----> 7 print(next(list_iter))

StopIteration: 
```

In [65]:

```python
mylist = ['Raj' , 'Vaibhav' , 'John' , 'Michael']
list_iter = iter(mylist) # Create an iterator object using iter()
print(list_iter.__next__()) # return first element in the iterator stream
print(list_iter.__next__()) # return next element in the iterator stream
print(list_iter.__next__())
print(list_iter.__next__())
```

```
Raj
Vaibhav
John
Michael
```

In [66]:

```python
mylist = ['Raj' , 'Vaibhav' , 'John' , 'Michael']
list_iter = iter(mylist) # Create an iterator object using iter()
for i in list_iter:
    print(i)
```

```
Raj
Vaibhav
John
Michael
```

In [67]:

```
# Looping Through an Iterable (tuple) using for Loop
mytuple = ('Raj' , 'Vaibhav' , 'John' , 'Michael')
for i in mytuple:
    print(i)
```

```
Raj
Vaibhav
John
Michael
```

In [68]:

```
# Looping Through an Iterable (string) using for Loop
mystr = "Hello Python"
for i in mystr:
    print(i)
```

```
H
e
l
l
o
 
P
y
t
h
o
n
```

In [69]:

```
class MyIter:
    def __init__(self):
        self.num = 0

    def __iter__(self):
        self.num = 1
        return self

    def __next__(self):
        if self.num <= 10:
            val = self.num
            self.num += 1
            return val
        else:
            raise StopIteration

# Creating an instance of MyIter
mynum = MyIter()
# Creating an iterator
iter1 = iter(mynum)

# Iterating over the iterator and printing values
for i in iter1:
    print(i)
```

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

In [70]:

```python
class MyIter:
    def __init__(self):
        self.num = 0

    def __iter__(self):
        self.num = 1
        return self

    def __next__(self):
        if self.num <= 20:
            val = self.num
            self.num += 2
            return val
        else:
            raise StopIteration

# Creating an instance of MyIter
my_odd = MyIter()
# Creating an iterator
iter1 = iter(my_odd)

# Iterating over the iterator and printing values
for i in iter1:
    print(i)
```

```
1
3
5
7
9
11
13
15
17
19
```

In [71]:

```
class MyFibonacci:
    def __init__(self):
        self.prev = 0
        self.cur = 0

    def __iter__(self):
        self.prev = 0
        self.cur = 1
        return self

    def __next__(self):
        if self.cur <= 50:
            val = self.cur
            self.cur += self.prev
            self.prev = val
            return val
        else:
            raise StopIteration

# Creating an instance of MyFibonacci
my_fibo = MyFibonacci()
# Creating an iterator
iter1 = iter(my_fibo)

# Iterating over the iterator and printing values
for i in iter1:
    print(i)
```

```
1
1
2
3
5
8
13
21
34
```

In [ ]: