UNIT 3 - Python OBJECT-ORIENTED

Definitions

- Class
  - A user-defined prototype for an object that defines a set of attributes that characterize any object of the class.
  - The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- Class variable
  - A variable that is shared by all instances of a class.
  - Class variables are defined within a class but outside any of the class's methods.
  - Class variables are not used as frequently as instance variables are.
- Data member
  - A class variable or instance variable that holds data associated with a class and its objects.
- Function overloading
  - The assignment of more than one behaviour to a particular function.
  - The operation performed varies by the types of objects or arguments involved.
- Instance variable
  - A variable that is defined inside a method and belongs only to the current instance of a class.
- Inheritance
  - The transfer of the characteristics of a class to other classes that are derived from it.
- Instance
  - An individual object of a certain class.
    - An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- Instantiation
  - The creation of an instance of a class.
- Method
  - A special kind of function that is defined in a class definition.
- Object
  - A unique instance of a data structure that's defined by its class.
  - An object comprises both data members (class variables and instance variables) and methods.
- Operator overloading
  - The assignment of more than one function to a particular operator.

Traits of OOPS

- Encapsulation
  - Mechanism that binds together code and data it manipulates and keeps both safe from outside interference and misuse.
- Polymorphism
  - Allows one interface to control access to a general class of actions.
- Inheritance
  - One object can acquire the properties of another object.

Creating a class

- The *class* statement creates a new class definition.
- The name of the class immediately follows the keyword *class* followed by a colon as follows

```
class ClassName:
        'Optional class documentation string’
        class_suite
```

- The class has a documentation string, which can be accessed via ClassName.__doc__.
- The class_suite consists of all the component statements defining class members, data attributes and functions.

```
class Lion(object):
      pass
```

- The (object) part in parentheses specifies the parent class that it is inheriting from.

Instance Attributes

- All classes create objects, and all objects contain characteristics called attributes.
- Use the __init__() method to initialize (e.g., specify) an object’s initial attributes by giving them their default value (or state).
- This method must have at least one argument as well as the self variable, which refers to the object itself

```
class Lion:
    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

- In the case of Lion() class, each lion has a specific name and age, which is obviously important to know for when actually creating different lions.
  - Remember: the class is just for defining the Lion, not actually creating instances of individual Lions with specific names and ages.
- The self variable is also an instance of the class.

- Since instances of a class have varying values
    - state Lion.name = name rather than self.name = name.
        - Since not all lions share the same name, the code should be able to assign different values to different instances.
        - Hence the need for the special self variable, which will help to keep track of individual instances of each class.

Class Attributes

- While instance attributes are specific to each object, class attributes are the same for all instances.
    - class Lion:
    -     # Class Attribute
    -     species = 'mammal'
    -     # Initializer / Instance Attributes
    -     def __init__(self, name, age):
    -         self.name = name
- So while each lion has a unique name and age, every lion will be a mammal.

Instantiating Objects

- Creating a new, unique instance of a class.
    - class Lion:
    -     pass
    - print(Lion())
    - print(Lion())
    - a = Lion()
    - b = Lion()
    - print(a==b)

Instance Methods

- Instance methods are defined inside a class and are used to get the contents of an instance.
- They can also be used to perform operations with the attributes of our objects.
- Like the __init__ method, the first argument is always self.
    - class Lion:
    -     # Class Attribute
    -     species = 'mammal'
    -     # Initializer / Instance Attributes
    -     def __init__(self, name, age):

- self.name = name
- self.age = age
- # instance method
- def description(self):
- return "{} is {} years old".format(self.name, self.age)
- # instance method
- def speak(self, sound):
- return "{} says {}".format(self.name, sound)

Built-in class functions

- Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute
  - __dict__
    - Dictionary containing the class's namespace.
  - __doc__
    - Class documentation string or none, if undefined.
  - __name__
    - Class name.
  - __module__
    - Module name in which the class is defined.
    - This attribute is "__main__" in interactive mode.
  - __bases__
    - A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

Creating a class - Example

```
class Employee:
'Common base class for all employees'
'empCount is a class variable whose value is shared among all instances of'
'this class'
'This can be accessed as Employee.empCount from inside the class or'
'outside the class.'
empCount = 0
'__init__() is a special method, which is called class constructor or'
'initialization method that Python calls when you create a new instance of'
'this class.'
def __init__(self, name, salary):
    self.name = name
```

```
        self.salary = salary
        Employee.empCount += 1
    'other class methods like normal functions with the exception that the first'
    'argument to each method is self’
    def displayCount(self):
        print ("Total Employee %d" % Employee.empCount)
```

Creating Instance Objects

- To create instances of a class, call the class using class name and pass in whatever arguments its __init__ method accepts.
- Example
  - emp1 = Employee("Zara", 2000)
  - emp2 = Employee("Manni", 5000)

Access to Attributes and Methods

- Access the object's attributes using the dot operator with object.
- Class variable would be accessed using class name as follows
- Example
  - emp1.displayEmployee()
  - emp2.displayEmployee()
  - print ("Total Employee %d" % Employee.empCount)

Constructor

- A constructor is a special type of method (function) which is used to initialize the instance members of the class.
- Constructor can be parameterized and non-parameterized as well.
- Constructor definition executes when the object is created for the class.
- Constructors also verify that there are enough resources for the object to perform any start-up task.

Constructor Creation

- A constructor is a class function that begins with double underscore (_).
- The name of the constructor is always the same __init__().
- While creating an object, a constructor can accept arguments if necessary.
- When a class is created without a constructor, Python automatically creates a default constructor that doesn't do anything.
- Every class must have a constructor, even if it simply relies on the default constructor.

Destructor

- A destructor is used to destroy the object and perform the final clean up.
- Although in python has a garbage collector to clean up the memory.
  - When an object is dereferenced or destroyed,
    - Closing open files
    - Closing database connections,
    - Cleaning up the buffer or cache
    - Freeing memory
- __del__ method is called for any object when the reference count for that object becomes zero.
- As reference counting is performed, hence it is not necessary that for an object __del__ method will be called if it goes out of scope.
- The destructor method will only be called when the reference count becomes zero.
- They are not called manually but completely automatic.

del keyword

- Any attribute of an object can be deleted anytime, using the del statement.
- The object itself can be deleted using the del statement.
- An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary).
- The object's reference count decreases when it's deleted with del, its reference is reassigned, or its reference goes out of scope.

Destroying Objects

- Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space.
- The process by which Python periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection.
- Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero.
- An object's reference count changes as the number of aliases that point to it changes.
- An object's reference count increases when it is assigned a new name or placed in a container (list, tuple, or dictionary).
- The object's reference count decreases when it's deleted with del, its reference is reassigned, or its reference goes out of scope.
- When an object's reference count reaches zero, Python collects it automatically.

self keywords

- The self in Python is equivalent to the this pointer in C++ and the this reference in Java and C#.
- self represents the instance of the class.
- By using the "self" keyword we can access the attributes and methods of the class in python.
- __init__ :
  - "__init__" is a reserved method in python classes.
  - It is known as a constructor in object-oriented concepts.
  - This method is called when an object is created from the class and it allows the class to initialize the attributes of a class.

Functions to access attributes

- getattr(obj, name[, default])
  - To access the attribute of object
- hasattr(obj,name)
  - To check if an attribute exists or not
- setattr(obj,name,value)
  - To set an attribute.
  - If attribute does not exist, then it would be created
- delattr(obj, name)
  - To delete an attribute

geattr attribute

- getattr() function is used to get the value of an object's attribute and if no attribute of that object is found, default value is returned.
- If any attribute that doesn't belong to the object has to be accessed, then the getattr() default value option can be used.
- It is easy to get the value by using the name of the attribute as String.
- If an attribute is not found some default value can be set, which enables to complete some of the incomplete data.
- If the class is work in progress, then use getattr() function to complete other code.
  - Once the class has this attribute, it will automatically pick it up and not use the default value.

hasattr attribute

- The hasattr(obj,name) − to check if an attribute exists or not.

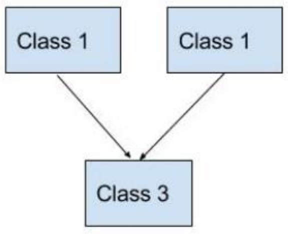
- The hasattr() function returns True if the specified object has the specified attribute, otherwise False.

delattr attribute

- The delattr() function will delete the specified attribute from the specified object.
  - delattr(object, attribute)

Inheritance

- Inheritance is the process by which one class takes on the attributes and methods of another.
- Newly formed classes are called child classes, and the classes that child classes are derived from are called parent classes.
- It’s important to note that child classes override or extend the functionality (e.g., attributes and behaviors) of parent classes.
- In other words, child classes inherit all of the parent’s attributes and behaviors but can also specify different behavior to follow.
- The most basic type of class is an object, which generally all other classes inherit as their parent.
- When a new class is defined, Python 3 it implicitly uses object as the parent class.
- The following two definitions are equivalent
  - class Lion(object):
  -     pass
  - # In Python 3, this is the same as:
  - class Lion:
  -     pass
- A class can be derived from more than one base classes in Python. This is called multiple inheritance.
- In multiple inheritance, the features of all the base classes are inherited into the derived class.

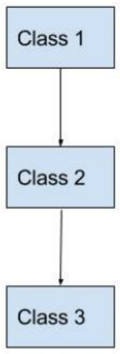


Inheritance – Multiple Inheritance

- In the multiple inheritance scenario, any specified attribute is searched first in the current class.
- If not found, the search continues into parent classes in depth-first, left-right fashion without searching same class twice.
  - This order is also called linearization of the multi derived class and the set of rules used to find this order is called Method Resolution Order (MRO).
- MRO must prevent local precedence ordering and also provide monotonicity.
- It ensures that a class always appears before its parents and in case of multiple parents, the order is same as tuple of base classes.
- MRO of a class can be viewed as the __mro__ attribute or mro() method.

Inheritance

- Multilevel inheritance is also possible in Python like other Object-Oriented programming languages.
- A derived class from another derived class, this process is known as multilevel inheritance.
- In Python, multilevel inheritance can be done at any depth.



- The isinstance() function is used to determine if an instance is also an instance of a certain parent class.

Encapsulation

- Encapsulation allows programmers better control of how data flows in their programs, and it protects that data.
- It also makes objects into more self-sufficient and independently functioning pieces.

Encapsulation – Access Modifiers

- Python has different levels of restriction that control how data can be accessed and from where.
- Variables and methods can be public, private, or protected.

- Those designations are made by the number of underscores before the variable or method.
- **Public**
  - Public variables and methods can be freely modified and run from anywhere, either inside or outside of the class.
  - To create a public variable or method, don't use any underscores.
- **Private**
  - The private designation only allows a variable or method to be accessed from within its own class or object.
  - The value of a private variable cannot be modified from outside of a class.
  - Private variables and methods are preceded by two underscores.
- **Protected**
  - Protected variables and methods are very similar to private ones.
  - A variable that is protected can only be accessed by its own class and any classes derived from it.
  - Protected variables begin with a single underscore.
- Encapsulation is the process of using private variables within classes to prevent unintentional or potentially malicious modification of data.
- By containing and protecting variables within a class, it allows the class and the objects that it creates to function as independent, self contained, parts functioning within the machine of the program itself.

Encapsulation - Setters and Getters

- The interfaces that are used for interacting with encapsulated variables are generally referred to as setter and getter methods because the are used to set and retrieve the values of variables.
- Because methods exist within a class or object, they are able to access and modify private variables, which cannot be done from outside the class.

Polymorphism

- Sometimes an object comes in many types or forms.
  - If there is a button, there are many different draw outputs (round button, check button, square button, button with image) but they do share the same logic: onClick().
- They all can be accessed using the same method.
- This idea is called *Polymorphism*.
- Polymorphism is based on the Greek words Poly (many) and morphism (forms).

- The polymorphism is the process of using an operator or function in different ways for different data input.
- In practical terms, polymorphism means that if class B inherits from class A, it doesn't have to inherit everything about class A; it can do some of the things that class A does differently.

Polymorphism - Examples

- Use the same indexing operator for three different data types.
- Use the same function in two different classes.

Operator Overloading

Operators	Methods
+	__add__(self, other)
–	__sub__(self, other)
*	__mul__(self, other)
//	__floordiv__(self, other)
/	__div__(self, other)
%	__mod__(self, other)
**	__pow__(self, other[, modulo])
<	__lt__(self, other)
<=	__le__(self, other)
==	__eq__(self, other)
!=	__ne__(self , other)
>=	__ge__(self, other)
&	__and__(self, other)
|	__or__(self, other)
^	__xor__(self, other)

Inner Classes

- An inner class or nested class is a defined entirely within the body of another class.
- If an object is created using a class, the object inside the root class can be used.
- A class can have more than one inner classes, but in general inner classes are avoided.
- An inner class can have both methods and variables.
- Example

  - create a class (Human) with one inner class (Head).
  - An instance is created that calls a method in the inner class

Factory Method

- The program may always know what kind of objects it has to create in advance.
- Some objects can be created only at execution time after a user requests so.
- Examples when you may use a factory method:
  - A user may click on a certain button that creates an object.
  - A user may create several new documents of different types.
  - If a user starts a browser, the browser does not know in advance how many tabs (where every tab is an object) will be opened.
- Use the factory method pattern.
- Have one function, the factory, that takes an input string and outputs an object.

Iterators

- Iterators are elegantly implemented within for loops, comprehensions, generators etc. but hidden in plain sight.
- Iterators are objects that can be iterated upon.
  - An object which will return data, one element at a time.
- Python iterator object must implement two special methods,
  - __iter__()
  - __next__()
    - collectively called the iterator protocol.
- Most of built-in containers in Python like: list, tuple, string etc. are iterables.
- The iter() function (which in turn calls the __iter__() method) returns an iterator from them.
- The next() function manually iterates through all the items of an iterator.
- When the end is reached and there is no more data to be returned, it will raise StopIteration.

Generators

- Generators are a simple way of creating iterators.
- Overheads in building an iterator in Python
  - Implement a class with __iter__() and __next__() method
  - Keep track of internal states
  - Raise StopIteration when there was no values to be returned.
- All the overhead mentioned above are automatically handled by generators in Python.

- A generator is a function that returns an object (iterator) which we can iterate over (one value at a time).
- If a function contains at least one yield statement (it may contain other yield or return statements), it becomes a generator function.
  - A return statement terminates a function entirely.
  - A yield statement pauses the function saving all its states and later continues from there on successive calls.

Generator function vs. Normal function

- Generator function contains one or more yield statement.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like __iter__() and __next__() are implemented automatically.
  - It can be iterated through the items using next().
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- When the function terminates, StopIteration is raised automatically on further calls.
- Unlike normal functions, the local variables are not destroyed when the function yields.
- The generator object can be iterated only once.
- To restart the process create another generator object.
  - Example → a = my_gen()
- Generators can be used with for loops directly.
- Generator functions are implemented with a loop having a suitable terminating condition.

Generator expression

- Generator expression creates an anonymous generator function.
- The syntax for generator expression is similar to that of a list comprehension in Python.
  - The square brackets are replaced with round parentheses.
- The major difference between a list comprehension and a generator expression is that while list comprehension produces the entire list, generator expression produces one item at a time.
- Generator expression can be used inside functions.
  - When used in such a way, the round parentheses can be dropped.

Closure - Preliminary

- A function defined inside another function is called a nested function.
- Nested functions can access variables of the enclosing scope.
- In Python, these non-local variables are read only by default and must declare them explicitly as non-local (using nonlocal keyword) in order to modify them.

Closure

- A closure is a combination of code and scope.
- The technique by which some data gets attached to the code is called closure in Python.
  - This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.
- A closure is a function object that remembers values in enclosing scopes regardless of whether those scopes are still present in memory.
- A closure is a function (object) that remembers its creation environment (enclosing scope).

When to have closure

- The criteria that must be met to create closure in Python are summarized in the following points.
  - There must be a nested function (function inside a function).
  - The nested function must refer to a value defined in the enclosing function.
  - The enclosing function must return the nested function.

When to use closure

- Closures can avoid the use of global values and provides some form of data hiding.
- It can also provide an object-oriented solution to the problem.
- When there are few methods (one method in most cases) to be implemented in a class, closures can provide an alternate and more elegant solutions.

When not to use closure

- When the number of attributes and methods get larger, better implement a class.

When are nested functions not a closure

- If nested functions don't
  - access variables that are local to enclosing scopes,
  - do so when they are executed outside of that scope,
- then they are not closures.

Decorators

- Functions and methods are called callable as they can be called.
- A decorator is a callable that returns a callable.
- A decorator takes in a function, adds some functionality and returns it.
- The decorator function added some new functionality to the original function.
  - It is similar to packing a gift.
- The decorator acts as a wrapper.
- The nature of the object that got decorated (actual gift inside) does not alter.
  - It only looks pretty (since it got decorated).
- Decorate a function and reassign it
  - variable = decorator(function)
  - variable()
- Decorators provide a simple syntax for calling higher-order functions.
- By definition, a decorator is a function that takes another function and extends the behaviour of the latter function without explicitly modifying it.

Decorators - Shortcut

- Use the @ symbol along with the name of the decorator function.
- Place it above the definition of the function to be decorated.
- Example

Shortcut	Equivalent

@make_pretty
def ordinary():
print("I am ordinary")
	def ordinary():
	print("I am ordinary")
	ordinary = make_pretty(ordinary)

Decorators – Preliminaries

- Everything in Python are objects.
- Names that are defined are simply identifiers bound to these objects.
- Various different names can be bound to the same function object.
- Functions can be passed as arguments to another function.
  - Functions like map, filter and reduce in Python
- Such function that take other functions as arguments are also called higher order functions.
- A function can return another function.
- Function decorators are simply wrappers to existing functions.

  - ∞ A function that takes another function as an argument, generates a new function, augmenting the work of the original function, and returning the generated function so that it can be used anywhere.
  - ∞ Example
    - ϖ To have get_text itself be decorated by p_decorate, assign get_text to the result of p_decorate
  - ∞ Decorated function takes a name argument.
- ₪ Methods are functions that expect their first parameter to be a reference to the current object.
- ₪ Decorators can be built for methods the same way, while taking self into consideration in the wrapper function.
- ₪ To make decorator useful for functions and methods alike.
  - ∞ Put *args and **kwargs as parameters for the wrapper, then it can accept any arbitrary number of arguments and keyword arguments.

***args and **kwargs**

- ₪ *args and **kwargs are mostly used in function definitions.
- ₪ *They allow to pass a variable number of arguments to a function.
  - ∞ These are conventions used.
  - ∞ It could have also be written as *var and **vars
- ₪ *args is used to send a non-keyworded variable length argument list to the function. (cannot be keywords)
- ₪ Any number and type of arguments can be passed to the list – number, string
- ₪ **kwargs allows to pass keyworded variable length of arguments to a function.
- ₪ **kwargs should be used if named arguments are to be handled in a function.
- ₪ Example
  - ∞ def my_three(a, b, c):
  - ∞ ……
  - ∞ a = {'a': "one", 'b': "two", 'c': "three" }
  - ∞ my_three(**a)
- ₪ Names of arguments in function must match with the name of keys in dictionary.
- ₪ Number of arguments should be same as number of keys in the dictionary.
- ₪ *args is used to help the developer passing a variable number of arguments to a function.
- ₪ The term variable is very important here as it basically is the key to understand the role of *args in the python programming language.
- ₪ The * unpacks the tuple elements

Regular Expressions

- Regular Expressions (sometimes shortened to regexp, regex, or re) are a tool for matching patterns in t
- Regular expression is a sequence of character(s) mainly used to find and replace patterns in a string or file.
- Regular expressions use two types of characters
  - Meta characters
    - These characters have a special meaning, similar to * in wild card.
  - Literals
    - Example a,b,1,2…
- Module "re" helps with regular expressions.
- Import library re before regular expressions can be used in Python.

Methods of Regular Expressions

- The 're' package provides multiple methods to perform queries on an input string.
- The most commonly used methods:
  - re.match()
  - re.search()
  - re.findall()
  - re.split()
  - re.sub()
  - re.compile()

re.match(pattern, string)

- This method finds match if it occurs at start of the string.
- Example
  - Calling match() on the string 'AV Analytics AV' and looking for a pattern 'AV' will match.
  - However, if we look for only Analytics, the pattern will not match.
- Can add methods like start() and end() to know the start and end position of matching pattern in the string.

re.search(pattern, string)

- It is similar to match() but it doesn't restrict us to find matches at the beginning of the string only.
- Unlike previous method, here searching for pattern 'Analytics' will return a match.

re.findall (pattern, string)

- It helps to get a list of all matching patterns.
- It has no constraints of searching from start or end.
- If we will use method findall to search a pattern in given string it will return all occurrences of the pattern.
- While searching a string, it is recommended to use re.findall() always, it can work like re.search() and re.match() both.

re.split(pattern, string, [maxsplit=0])

- This methods helps to split string by the occurrences of given pattern.
- Method split() has another argument "maxsplit".
  - It has default value of zero.
  - In this case it does the maximum splits that can be done, but if we give value to maxsplit, it will split the string.

re.sub(pattern, repl, string)

- It helps to search a pattern and replace with a new sub string.
- If the pattern is not found, *string* is returned unchanged.

re.compile(pattern, repl, string)

- Combine a regular expression pattern into pattern objects, which can be used for pattern matching.
- It also helps to search a pattern again without rewriting it.

Operators used in Regular Expressions

Operator	Description
.	Matches with any single character except newline '\n'.
?	Match 0 or 1 occurrence of the pattern to its left
+	1 or more occurrences of the pattern to its left
0	0 or more occurrences of the pattern to its left
\w	Matches with an alphanumeric character whereas
\d	Matches with digits [0-9]
\W	Matches non-alphanumeric character
/D	Matches with non-digits
\s	Matches with a single white space character (space, newline, return, tab, form)

\S	Matches any non-white space character.
\b	boundary between word and non-word
/B	Opposite of /b
[..]	Matches any single character in a square bracket
[^..]	Matches any single character not in square bracket
\	It is used for special meaning characters
\.	to match a period
\+	To match for plus sign
^ and $	^ and $ match the start or end of the string respectively
{n,m}	Matches at least n and at most m occurrences of preceding expression if we write it as
{,m}	It will return at least any minimum occurrence to max m preceding expression.
a|b	Matches either a or b
( )	Groups regular expressions and returns matched text
\t, \n, \r	Matches tab, newline, return

Regular Expression Modifiers – Option Flags

Modifier	Description
re.I	Performs case-insensitive matching.
re.L	Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior (\b and \B).
re.M	Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
re.S	Makes a period (dot) match any character, including a newline.
re.U	Interprets letters according to the Unicode character set. This flag affects the behaviour of \w, \W, \b, \B.
re.X	Permits "cuter" regular expression syntax.

	It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.
re.I	Performs case-insensitive matching.

Methods

```
>>> class MyClass:
    def method(self) -> object:
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'


>>> obj = MyClass()
>>> obj.method()
('instance method called', <__main__.MyClass object at 0x0000021E43C62E80>)
>>> MyClass.method(obj)
('instance method called', <__main__.MyClass object at 0x0000021E43C62E80>)
>>> obj.classmethod()
('class method called', <class '__main__.MyClass'>)
```

- The first method on MyClass, called method, is a regular instance method.
  - Through the self parameter, instance methods can freely access attributes and other methods on the same object.
- The second is a class method.
  - Instead of accepting a self parameter, class methods take a cls parameter that points to the class—and not the object instance—when the method is called.
  - Because the class method only has access to this cls argument, it can't modify object instance state.
- The third method, MyClass.staticmethod was marked with a @staticmethod decorator to flag it as a static method.
  - This type of method takes neither a self nor a cls parameter.
  - A static method can neither modify object state nor class state.
  - Static methods are restricted in what data they can.

```
>>> MyClass.classmethod()
('class method called', <class '__main__.MyClass'>)
>>> MyClass.staticmethod()
'static method called'
>>> MyClass.method()
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    MyClass.method()
TypeError: method() missing 1 required positional argument: 'self'
```

Programs

#!/usr/bin/python	class Student:
class BankAccount:	# Constructor - parameterized
def __init__(self):	def __init__(self, name):
self.balance = 0	print("This is parametrized constructor")
def withdraw(self, amount):	self.name = name
self.balance -= amount	def show(self):
if self.balance < 0 :	print("Hello",self.name)
print("Withdrawl not possible")	student = Student("Isha")
self.balance += amount	student.show()
return self.balance	
def deposit(self, amount):	class Student:
self.balance += amount	# Constructor – non-parameterized
return self.balance	def __init__(self):
def display(self):	print("This is non parametrized constructor")
print(self.balance)	def show(self,name):
a = BankAccount()	print("Hello",name)
a.display()	student = Student()
d=int(input("Enter the amount to be deposited:"))	student.show("Isha")
a.deposit(d)	
a.display()	class Point:
w = int(input("Enter the amount to be withdrawn:"))	def __init__(self, x = 0, y = 0):
a.withdraw(w)	self.x = x
a.display()	self.y = y
b = BankAccount()	def __str__(self):
b.display()	return "({0},{1})".format(self.x, self.y)
d=int(input("Enter the amount to be deposited:"))	def __add__(self, other):
b.deposit(d)	x = self.x + other.x
b.display()	y = self.y + other.y

w = int(input("Enter the amount to be withdrawn:"))	return Point(x, y)
b.withdraw(w)	def __lt__(self, other):
b.display()	self_mag = (self.x ** 2) + (self.y ** 2)
a.display()	other_mag = (other.x ** 2) + (other.y ** 2)
	return self_mag < other_mag
class ComplexNumber:	p1 = Point(4,-3)
def __init__(self, r=0, i=0):	p2 = Point(-1,2)
self.real = r	print(p1 + p2)
self.imag = i	print(Point(1,1) < Point(-2,-3))
def getData(self):	
print("{0}+{1}j".format(self.real, self.imag))	class PowTwo:
n = int(input("Enter the number:"))	"""Class to implement an iterator of powers of two"""
m = int(input("Enter the number:"))	def __init__(self, max=0):
c=ComplexNumber(n,m)	self.max = max
c.getData()	def __iter__(self):
	self.n = 0
#!/usr/bin/python	return self
# define a list	def __next__(self):
my_list = [4, 7, 0, 3]	if self.n <= self.max:
# get an iterator using iter()	result = 2 ** self.n
my_iter = iter(my_list)	self.n += 1
## iterate through it using next()	return result
print(next(my_iter))	else:
print(next(my_iter))	raise StopIteration
print(my_iter.__next__())	for i in PowTwo(5):
print(my_iter.__next__())	print(i)
next(my_iter)	
	class InfIter:
x = iter([1, 2, 3])	"""Infinite iterator to return all odd numbers"""
print(x.__next__())	def __iter__(self):

print(x.__next__())	self.num = 1
print(x.__next__())	return self
print(x.__next__())	def __next__(self):
	num = self.num
	self.num += 2
	return num
	for i in InfIter():
	print(i)

import re	result=re.findall(r'.','AV is largest Analytics community of India')
result = re.match(r'AV', 'AV Analytics Vidhya AV')	print(result)
'Use “r” at the start of the pattern string, it designates a python raw string.'	result=re.findall(r'\w','AV is largest Analytics community of India')
print(result)	print(result)
print(result.group(0))	result=re.findall(r'\w*','AV is largest Analytics community of India')
result = re.match(r'Analytics', 'AV Analytics Vidhya AV')	print(result)
print(result)	result=re.findall(r'\w+','AV is largest Analytics community of India')
result = re.match(r'AV', 'AV Analytics Vidhya AV')	print(result)
print(result.start())	result=re.findall(r'^\w+','AV is largest Analytics community of India')
print(result.end())	print(result)
result = re.search(r'Analytics', 'AV Analytics Vidhya AV')	result=re.findall(r'\w+$','AV is largest Analytics community of India')
print(result.group(0))	print(result)
result = re.findall(r'AV', 'AV Analytics Vidhya AV')	result=re.findall(r'\w\w','AV is largest Analytics community of India')
print(result)	print(result)

result=re.split(r'y','Analytics')	result=re.findall(r'\b\w.','AV is largest Analytics community of India')
print(result)	print(result)
result=re.split(r'i','Analytics Vidhya')	result=re.findall(r'\B\w.','AV is largest Analytics community of India')
print(result)	print(result)
	result=re.findall(r'@\w+','abc.test@gmail.com, xyz@test.in, test.first@analyticsvidhya.com, first.test@rest.biz')
	print(result)
	result=re.findall(r'@\w+.\w+','abc.test@gmail.com, xyz@test.in, test.first@analyticsvidhya.com, first.test@rest.biz')
	print(result)
	result=re.findall(r'@\w+.(\w+)','abc.test@gmail.com, xyz@test.in, test.first@analyticsvidhya.com, first.test@rest.biz')
	print(result)
	result=re.findall(r'\d{2}-\d{2}-\d{4}','Amit 34-3456 12-05-2007, XYZ 56-4532 11-11-2011, ABC 67-8945 12-01-2009')
	print(result)
	result=re.findall(r'\d{2}-\d{2}-(\d{4})','Amit 34-3456 12-05-2007, XYZ 56-4532 11-11-2011, ABC 67-8945 12-01-2009')
	print(result)
	result=re.findall(r'\b[aeiouAEIOU]\w+','AV is largest Analytics community of India')
	print(result)
	result=re.findall(r'\b[^aeiouAEIOU]\w+','AV is largest Analytics community of India')
	print(result)
	li=['9999999999','999999-999','99999x9999']

	for val in li:
	if re.match(r'[8-9]{1}[0-9]{9}',val) and len(val) == 10:
	print('yes')
	else:
	print('no')
	line = 'asdf fjdk;afed,fjek,,asdf,foo.llll' # String has multiple delimiters (";",","," ").
	result= re.split(r'[;,.\s]', line)
	print(result)
	result= re.sub(r'[;,.\s]',' ', line)
	print(result)
	result=re.findall(r'.','AV is largest Analytics community of India')
	print(result)