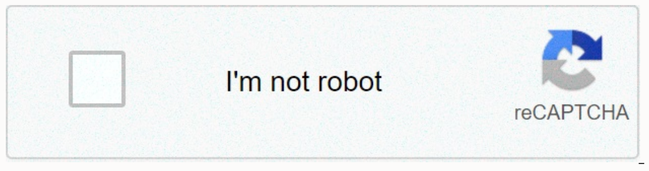

Continue

Object oriented programming concepts python pdf

Python is an object-oriented programming language. What this means is we can solve a problem in Python by creating objects in our programs. In this guide, we will discuss OOPs terms such as class, objects, methods etc. along with the Object oriented programming features such as inheritance, polymorphism, abstraction, encapsulation. Object An object is an entity that has attributes and behaviour. For example, Ram is an object who has attributes such as height, weight, color etc. and has certain behaviours such as walking, talking, eating etc. Class A class is a blueprint for the objects. For example, Ram, Shyam, Steve, Rick are all objects so we can define a template (blueprint) class Human for these objects. The class can define the common attributes and behaviours of all the objects. Methods As we discussed above, an object has attributes and behaviours. These behaviours are called methods in programming. Example of Class and Objects In this example, we have two objects Ram and Steve that belong to the class Human Object attributes: name, height, weight Object behaviour: eating() Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are downright easy. This chapter helps you become an expert in using Python's object-oriented programming support. If you do not have any previous experience with object-oriented (OO) programming, you may want to consult an introductory course on it or at least a tutorial of some sort so that you have a grasp of the basic concepts. However, here is small introduction of Object-Oriented Programming (OOP) to bring you at speed − Overview of OOP Terminology 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 behavior 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. Creating Classes 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. Example Following is the example of a simple Python class − class Employee: 'Common base class for all employees' empCount = 0 def __init__(self, name, salary): self.name = name self.salary = salary Employee.empCount += 1 def displayCount(self): print "Total Employee %d" % Employee.empCount def displayEmployee(self): print "Name : ", self.name, ", Salary: ", self.salary The variable empCount is a class variable whose value is shared among all instances of a this class. This can be accessed as Employee.empCount from inside the class or outside the class. The first method __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. You declare other class methods like normal functions with the exception that the first argument to each method is self. Python adds the self argument to the list for you; you do not need to include it when you call the methods. Creating Instance Objects To create instances of a class, you call the class using class name and pass in whatever arguments its __init__ method accepts. "This would create first object of Employee class" emp1 = Employee("Zara", 2000) "This would create second object of Employee class" emp2 = Employee("Manni", 5000) Accessing Attributes You access the object's attributes using the dot operator with object. Class variable would be accessed using class name as follows − emp1.displayEmployee() emp2.displayEmployee() print "Total Employee %d" % Employee.empCount Now, putting all the concepts together − #!/usr/bin/python class Employee: 'Common base class for all employees' empCount = 0 def __init__(self, name, salary): self.name = name self.salary = salary Employee.empCount += 1 def displayCount(self): print "Total Employee %d" % Employee.empCount def displayEmployee(self): print "Name : ", self.name, ", Salary: ", self.salary "This would create first object of Employee class" emp1 = Employee("Zara", 2000) "This would create second object of Employee class" emp2 = Employee("Manni", 5000) emp1.displayEmployee() emp2.displayEmployee() print "Total Employee %d" % Employee.empCount When the above code is executed, it produces the following result − Name : Zara ,Salary: 2000 Name : Manni ,Salary: 5000 Total Employee 2 You can add, remove, or modify attributes of classes and objects at any time − emp1.age = 7 # Add an 'age' attribute. emp1.age = 8 # Modify 'age' attribute. del emp1.age # Delete 'age' attribute. Instead of using the normal statements to access attributes, you can use the following functions − The getattr(obj, name[, default]) − to access the attribute of object. The hasattr(obj,name) − to check if an attribute exists or not. The setattr(obj,name,value) − to set an attribute. If attribute does not exist, then it would be created. The delattr(obj, name) − to delete an attribute. hasattr(emp1, 'age') # Returns true if 'age' attribute exists getattr(emp1, 'age') # Returns value of 'age' attribute setattr(emp1, 'age', 8) # Set attribute 'age' at 8 delattr(empl, 'age') # Delete attribute 'age' Built-In Class Attributes 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. For the above class let us try to access all these attributes − #!/usr/bin/python class Employee: 'Common base class for all employees' empCount = 0 def __init__(self, name, salary): self.name = name self.salary = salary Employee.empCount += 1 def displayCount(self): print "Total Employee %d" % Employee.empCount def displayEmployee(self): print "Name : ", self.name, ", Salary: ", self.salary print "Employee.__doc__:", Employee.__doc__ print "Employee.__name__:", Employee.__name__ print "Employee.__module__:", Employee.__module__ print "Employee.__bases__:", Employee.__bases__ print "Employee.__dict__:", Employee.__dict__ When the above code is executed, it produces the following result − Employee.__doc__: Common base class for all employees Employee.__name__: Employee Employee.__module__: __main__ Employee.__bases__: () Employee.__dict__: {'__module__': '__main__', 'displayCount': , 'empCount': 2, 'displayEmployee': , '__doc__': 'Common base class for all employees', '__init__': } Destroying Objects (Garbage Collection) 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. a = 40 # Create object b = a # Increase ref. count of c = [b] # Increase ref. count of del a # Decrease ref. count of b = 100 # Decrease ref. count of c[0] = -1 # Decrease ref. count of You normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space. But a class can implement the special method __del__(), called a destructor, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non memory resources used by an instance. Example This __del__() destructor prints the class name of an instance that is about to be destroyed − #!/usr/bin/python class Point: def __init__(self, x=0, y=0): self.x = x self.y = y def __del__(self): class_name = self.__class__.__name__ print class_name, "destroyed" pt1 = Point() pt2 = pt1 pt3 = pt1 print id(pt1), id(pt2), id(pt3) # prints the ids of the obejcts del pt1 del pt2 del pt3 When the above code is executed, it produces following result − 3083401324 3083401324 3083401324 Point destroyed Note − Ideally, you should define your classes in separate file, then you should import them in your main program file using import statement. Class Inheritance Instead of starting from scratch, you can create a class by deriving it from a preexisting class by listing the parent class in parentheses after the new class name. The child class inherits the attributes of its parent class, and you can use those attributes as if they were defined in the child class. A child class can also override data members and methods from the parent. Syntax Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name − class SubClassName (ParentClass1[, ParentClass2, ...]): 'Optional class documentation string' class_suite Example #!/usr/bin/python class Parent: # define parent class parentAttr = 100 def __init__(self): print "Calling parent constructor" def parentMethod(self): print 'Calling parent method' def setAttr(self, attr): Parent.parentAttr = attr def getAttr(self): print "Parent attribute :", Parent.parentAttr class Child(Parent): # define child class def __init__(self): print "Calling child constructor" def childMethod(self): print 'Calling child method' c = Child() # instance of child c.childMethod() # child calls its method c.parentMethod() # calls parent's method c.setAttr(200) # again call parent's method c.getAttr() # again call parent's method When the above code is executed, it produces the following result − Calling child constructor Calling child method Calling parent method Parent attribute : 200 Similar way, you can drive a class from multiple parent classes as follows − class A: # define your class A ..... class B: # define your class B ..... class C(A, B): # subclass of A and B ..... You can use issubclass() or isinstance() functions to check a relationships of two classes and instances. The issubclass(sub, sup) boolean function returns true if the given subclass sub is indeed a subclass of the superclass sup. The isinstance(obj, Class) boolean function returns true if obj is an instance of class Class or is an instance of a subclass of Class Overriding Methods You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass. Example #!/usr/bin/python class Parent: # define parent class def myMethod(self): print 'Calling parent method' class Child(Parent): # define child class def myMethod(self): print 'Calling child method' c = Child() # instance of child c.myMethod() # child calls overridden method When the above code is executed, it produces the following result − Calling child method Base Overloading Methods Following table lists some generic functionality that you can override in your own classes − Sr.No. Method, Description & Sample Call 1 __init__ (self [,args...]) Constructor (with any optional arguments) Sample Call : obj = className(args) 2 __del__(self) Destructor, deletes an object Sample Call : del obj 3 __repr__(self) Evaluable string representation Sample Call : repr(obj) 4 __str__(self) Printable string representation Sample Call : str(obj) 5 __cmp__ (self, x) Object comparison Sample Call : cmp(obj, x) Overloading Operators Suppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you. You could, however, define the __add__ method in your class to perform vector addition and then the plus operator would behave as per expectation − Example #!/usr/bin/python class Vector: def __init__(self, a, b): self.a = a self.b = b def __str__(self): return 'Vector (%d, %d)' % (self.a, self.b) def __add__(self,other): return Vector(self.a + other.a, self.b + other.b) v1 = Vector(2,10) v2 = Vector(5,-2) print v1 + v2 When the above code is executed, it produces the following result − Vector(7,8) Data Hiding An object's attributes may or may not be visible outside the class definition. You need to name attributes with a double underscore prefix, and those attributes then are not be directly visible to outsiders. Example #!/usr/bin/python class JustCounter: __secretCount = 0 def count(self): self.__secretCount += 1 print self.__secretCount counter = JustCounter() counter.count() counter.count() print counter.__secretCount When the above code is executed, it produces the following result − 1 2 Traceback (most recent call last): File "test.py", line 12, in print counter.__secretCount AttributeError: JustCounter instance has no attribute '__secretCount' Python protects those members by internally changing the name to include the class name. You can access such attributes as object._className__attrName. If you would replace your last line as following, then it works for you − ..... print counter._JustCounter__secretCount When the above code is executed, it produces the following result − 1 2 2