OO Part 3 - Python

|  |  |
| --- | --- |
| **OO in Python**   * Provides object orientation via class definitions which can be used to instantiate objects. * Simple syntax changes to support OO * Allows multiple inheritance classes; therefore, interfaces are not needed. * No specification of private vs public * Supports class variables, class methods, instance variables and instance methods. * Provides introspection methods to access metadata | from Employee import \*  from Manager import \*  emp = Employee("Rob Benz", 125000, "Dr")  print("Name: %s, Pay: %.2f"%(emp.getFormattedName(), emp.calculatePay()))  emp = Employee("Mae Nard", 75000)  print("Name: %s, Pay: %.2f"%(emp.getFormattedName(),  emp.calculatePay())) |
| **Defining a Class - Python**  class *ClassName*:  'class documentation'  *assignValuesToClassVariables*  def \_\_init\_\_(self, c*onstructorParameters*):  *assignValuesToInstanceVariables*  Instead of **multiple constructors**, use **optional arguments**.  **Class variables** are assigned values outside the methods to initialize them. They can then be referenced inside methods using:  *ClassName*.*classVariableName*  **Instance variables** are created in the constructor method named \_\_init\_\_ . They are referenced by using:  self.*instanceVariableName*  **Class methods** are created by preceding them with **@classmethod**. The method is passed a class type object; therefore, it must receive that as its first argument.  @classmethod  def *classMethodName*(cls, *classMethodParameters*):  *details*  **Instance methods** are created by **def** statements. The method is passed an instance as its first argument.  def *instanceMethodName*(self, *methodParameters*):  *details*  **Class methods can be invoked**  by specifying:  *Classname.methodName*(*parameters*) | # Employee.py  class Employee:  'Employee superclass for all employees'  lastId = 0 # class variable  # define constructor (notice the use of  # optional parameters instead of multiple  # constructors)  def \_\_init\_\_(self, name, salary, prefix="", exempt=1):  # define instance variables  self.namePrefix = prefix  self.name = name  self.salary = salary  self.exemptions = exempt  self.id = Employee.generateID()    #define class methods  @classmethod  def generateID(cls):  if Employee.lastId == 0:  Employee.lastId = 111 #db.getLastId()  Employee.lastId += 1  return Employee.lastId    #define instance methods  def getFormattedName(self):  if self.namePrefix == "":  return self.name  else:  return self.namePrefix + " " + self.name  def calculatePay(self):  if self.exemptions == 1:  return self.salary / 12  else:  return self.salary / 12 \* 0.9 |
| **Important Concepts - Inheritance**  To define a subclass, use the following:  class *SubClassName*(*SuperClassName*):  To invoke the superclass' constructor in the subclass, use the following:  super().\_\_init\_\_(*superClassParameters*)  To invoke an instance method in the superclass:  super().*methodName*(*parameters*) | # Manager.py  from Employee import \*  class Manager(Employee):  def \_\_init\_\_(self, name, salary, bonus  , prefix="", exempt=1):  self.bonus = bonus  super().\_\_init\_\_(name, salary, prefix=prefix, exempt=exempt)    def calculatePay(self):  return super().calculatePay() + self.bonus/12 |
| **Important Concepts - Polymorphism** | # EmployeeOO.py  from Employee import \*  from Manager import \*  emp = Employee("Rob Benz", 125000, "Dr")  print("Name: %s, Pay: %.2f"%(emp.getFormattedName(), emp.calculatePay()))  emp = Employee("Mae Nard", 75000)  print("Name: %s, Pay: %.2f"%(emp.getFormattedName(), emp.calculatePay()))  emp = Manager("R Pepper", 150000, 10000, prefix="Dr", exempt=2)  print("Name: %s, Pay: %.2f"%(emp.getFormattedName(),  emp.calculatePay())) |
| **Important Concepts - Encapsulation** | # StackOne.py without raising exception  class StackOne:  def \_\_init\_\_(self):  self.itemM = []  def push(self, item):  self.itemM.append(item)  def pop(self):  return self.itemM.pop()  def isEmpty(self):  return self.itemM == []  def size(self):  return len(self.itemM) |
| The example on the right uses a linked list to implement a stack. The Node class is a nested class known to StackTwo.  When creating an instance of the Node class, we also reference the surrounding class.  # main  from StackTwo import \*  stack = StackTwo()  stack.push(10)  stack.push(20)  print(stack.size(), stack.isEmpty())  print (stack.pop())  print (stack.pop())  print (stack.pop())  **Output:**  2 False  20  10  Traceback (most recent call last):  File "main.py", line 9, in <module>  print (stack.pop())  File "/home/clark/cs3723/Python/StackOO/StackTwo.py", line 18, in pop  item = self.top.getData()  AttributeError: 'NoneType' object has no attribute 'getData' | # StackTwo.py using a linked list  # The Node class is nested.  class StackTwo:  class Node:  def \_\_init\_\_(self,data,next):  self.data = data  self.next = next  def getData(self):  return self.data  def getNext(self):  return self.next  def \_\_init\_\_(self):  self.top = None  self.count = 0  def push(self, item):  self.top = StackTwo.Node(item,self.top)  self.count += 1  def pop (self):  item = self.top.getData()  self.top = self.top.getNext()  self.count -= 1  return item  def isEmpty(self):  return self.top == None  def size(self):  return self.count |
| **Exception Classes**  To declare an exception class, subclass it from another Exception class. In this example, we passed all of its arguments to the super class.  Since the prevous example raised an AttributeError due to self.top being None and a reference to self.data in getData(), we will consume that exception in StackTwo's pop() and raise a StackUnderFlow exception.  # main  from StackTwov2 import \*  stack = StackTwo()  stack.push(10)  stack.push(20)  print(stack.size(), stack.isEmpty())  print (stack.pop())  print (stack.pop())  print (stack.pop())  **Output:**  2 False  20  10  Traceback (most recent call last):  File "/home/clark/cs3723/Python/StackOO/StackTwov2.py", line 24, in pop  item = self.top.getData()  AttributeError: 'NoneType' object has no attribute 'getData'  During handling of the above exception, another exception occurred:  Traceback (most recent call last):  File "mainv2wo.py", line 9, in <module>  print (stack.pop())  File "/home/clark/cs3723/Python/StackOO/StackTwov2.py", line 28, in pop  raise StackUnderFlow | # StackTwov2.py  class StackUnderFlow(Exception):  def \_\_init\_\_(self, \*args, \*\*kwargs):  super().\_\_init\_\_(self, \*args, \*\*kwargs)  class StackTwo:  class Node:  def \_\_init\_\_(self,data,next):  self.data = data  self.next = next  def getData(self):  return self.data  def getNext(self):  return self.next  def \_\_init\_\_(self):  self.top = None  self.count = 0  def push(self, item):  self.top = StackTwo.Node(item,self.top)  self.count += 1  def pop (self):  try:  item = self.top.getData()  self.top = self.top.getNext()  self.count -= 1  except AttributeError:  raise StackUnderFlow  return item  def isEmpty(self):  return self.top == None  def size(self):  return self.count |
| **We can change the invoking code to handle the exception** | # main  from StackTwov2 import \*  stack = StackTwo()  stack.push(10)  stack.push(20)  print(stack.size(), stack.isEmpty())  try:  print (stack.pop())  print (stack.pop())  print (stack.pop())  except StackUnderFlow:  print("underflow")  **Output:**  2 False  20  10  underflow |
| **checking class type**  Python provides two built-in functions for checking the type of an object:  **isinstance**(*obj*, *class*)  Returns True if the instance, *obj*, is either an instance of the class or a subclass of the specified class.  **issubclass**(*class*, *superClass*)  Returns True if the specified class is either the specified *superClass* or a subclass of the specified *superClass.*  You can also use **type**(*obj*) which returns the type of the object. | # EmployeeOOType.py  from Employee import \*  from Manager import \*  emp = Employee("Mae Nard", 75000)  print("Name: %s, Pay: %.2f"%(emp.getFormattedName()  , emp.calculatePay()))  print("isinstance", isinstance(emp, Employee)  , isinstance(emp, Manager), type(emp))  if (type(emp) == Employee):  print("%s is an Employee"% emp.getFormattedName());  else:  print("%s is a Manager"% emp.getFormattedName())  emp = Manager("R Pepper", 150000, 10000, prefix="Dr", exempt=2)  print("Name: %s, Pay: %.2f"%(emp.getFormattedName()  , emp.calculatePay()))  print("isinstance", isinstance(emp, Employee)  , isinstance(emp, Manager), type(emp))  if (type(emp) == Employee):  print("%s is an Employee"% emp.getFormattedName());  else:  print("%s is a Manager"% emp.getFormattedName())  print("issubclass", issubclass(Employee, Employee)  , issubclass(Manager, Employee))  **Output:**  **Name: Mae Nard, Pay: 6250.00**  **isinstance True False <class 'Employee.Employee'>**  **Mae Nard is an Employee**  **Name: Dr R Pepper, Pay: 12083.33**  **isinstance True True <class 'Manager.Manager'>**  **Dr R Pepper is a Manager**  **issubclass True True** |
| **metadata, built-in attribute functions**  Python provides the following functions to get metadata about an object:  **type**(*obj*)  Returns the class for the specified instance. See the example above.  **hasattr**(*obj*, *attributeName*)  Returns True if the instance has the specified attribute.  **dir**(*obj*)  Returns a list of the attributes and methods for the specified object. This includes the "\_\_*name\_\_*" Python reserved names.  *obj***.\_\_dict\_\_**  Returns a dictionary of the attributes and values.  *obj***.\_\_class\_\_.\_\_name\_\_**  Returns just the class name..  The following methods can be used to get, set, and delete attributes on an object:  **getattr**(*obj, attributeName*)  *attributeName - string for the name of an attribute*  Returns the value of the specified attributeName for the specified instance. You can also specify an optional third argument for the default value if the attribute doesn't exist.  If the specified *attributeName* is a **function name**, it returns a closure method which can be used in a dynamic call.  **setattr**(*obj*, *attributeName, value*)  Sets the specified attribute to the specified value.  **delattr**(*obj*, *attributeName*)  Deletes the specified attribute to the specified value. | # EmployeeOOMeta.py  from Employee import \*  from Manager import \*  emp = Employee("Mae Nard", 75000)  print("\*\*\* Name: %s, Pay: %.2f"%(emp.getFormattedName()  , emp.calculatePay()))  # print the class name and the type  print(">>>", emp.\_\_class\_\_.\_\_name\_\_)  print("type is", type(emp))  if hasattr(emp, "salary"):  print("salary=", getattr(emp, "salary", None))  if hasattr(emp, "bonus"):  print("bonus=", getattr(emp, "bonus", None))  print(">>> printing dir(emp)")  print (dir(emp))  # Dynamically call calculatePay  f = getattr(emp, "calculatePay", None)  if f != None:  print("calculatePay exists, function:", f)  print("dynamic call to calculatePay:", f())  else:  print("no calculatePay")  print(" ")  emp = Manager("R Pepper", 150000, 10000, prefix="Dr", exempt=2)  print("\*\*\* Name: %s, Pay: %.2f"%(emp.getFormattedName()  , emp.calculatePay()))  # print the class name and the type  print(">>>", emp.\_\_class\_\_.\_\_name\_\_)  print("type is", type(emp))  if hasattr(emp, "salary"):  print("salary=", getattr(emp, "salary", None))  if hasattr(emp, "bonus"):  print("bonus=", getattr(emp, "bonus", None))  print(">>> printing dir(emp)")  print (dir(emp))  # Dynamically call calculatePay  f = getattr(emp, "calculatePay", None)  if f != None:  print("calculatePay exists, function:", f)  print("dynamic call to calculatePay:", f())  else:  print("no calculatePay") |
| Output from the example above. | **Output:**  **\*\*\* Name: Mae Nard, Pay: 6250.00**  **>>> Employee**  **type is <class 'Employee.Employee'>**  **salary= 75000**  **>>> printing dir(emp)**  **['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_h**  **ash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_'**  **, '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'calculatePay', 'exemptions', 'generateID', 'getFormattedName', 'id',**  **'lastId', 'name', 'namePrefix', 'salary']**  **calculatePay exists, function: <bound method Employee.calculatePay of <Employee.Employee object at 0x00000000031FD2E8>>**  **dynamic call to calculatePay: 6250.0**  **\*\*\* Name: Dr R Pepper, Pay: 12083.33**  **>>> Manager**  **type is <class 'Manager.Manager'>**  **salary= 150000**  **bonus= 10000**  **>>> printing dir(emp)**  **['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_h**  **ash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_'**  **, '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'bonus', 'calculatePay', 'exemptions', 'generateID', 'getFormattedNam**  **e', 'id', 'lastId', 'name', 'namePrefix', 'salary']**  **calculatePay exists, function: <bound method Manager.calculatePay of <Manager.Manager object at 0x00000000031FD320>>**  **dynamic call to calculatePay: 12083.333333333334** |
| **Simple printObj utility**  # PrtObj.py  def printObj(obj):  attrD = obj.\_\_dict\_\_  print(">>>", obj.\_\_class\_\_.\_\_name\_\_)  for attrNm, attrVal in attrD.items():  print(attrNm,"=",attrVal,sep="") | # The following code uses the PrtObj utility:  from Employee import \*  from Manager import \*  from PrtObj import printObj  emp = Employee("Mae Nard", 75000)  printObj(emp)  emp = Manager("R Pepper", 150000, 10000, prefix="Dr"  , exempt=2)  printObj(emp)  **Output:**  **>>> Employee**  **salary=75000**  **exemptions=1**  **name=Mae Nard**  **id=112**  **namePrefix=**  **>>> Manager**  **salary=150000**  **exemptions=2**  **name=R Pepper**  **id=113**  **namePrefix=Dr**  **bonus=10000** |