**EXPERIMENT # 03**

**INTRODUCTION**

**The purpose of this lab is to get familiar with object oriented concept in Python**

|  |  |
| --- | --- |
| **Name** | **Hadeer-Ur-Rehman**  **S.M Yawar Abbas** |
| **Date** | **8/12/2020** |
| **Registration No** | **12413**  **13651** |
| **Department** | **Computer Science** |
| **Total Marks** |  |
| **Marks Obtained** |  |
| **Remarks** |  |
|  |
|  |

# \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lab Instructor Signature**

|  |  |
| --- | --- |
| **INTRODUCTION** | **Experiment** 03 |

**OBJECTIVE**

You will learn syntax and semantics of advanced concepts of Python, and get introduced to structured data objects.

# THEORY

## Object Basics

Although this isn't a class in object-oriented programming, you'll have to use some objects in the programming projects, and so it's worth covering the basics of objects in Python. An object encapsulates data and provides functions for interacting with that data.

## Defining Classes

Here's an example of defining a class named FruitShop: class FruitShop:

def \_\_init\_\_(self, name, fruitPrices):

"""

name: Name of the fruit shop

fruitPrices: Dictionary with keys as fruit strings and prices for values e.g.

{'apples':2.00, 'oranges': 1.50, 'pears': 1.75}

"""

self.fruitPrices = fruitPrices self.name = name

print('Welcome to %s fruit shop' % (name))

def getCostPerPound(self, fruit):

"""

fruit: Fruit string

Returns cost of 'fruit', assuming 'fruit' is in our inventory or None otherwise

""" if fruit not in self.fruitPrices: return None

return self.fruitPrices[fruit]

def getPriceOfOrder(self, orderList):

"""

orderList: List of (fruit, numPounds) tuples

Returns cost of orderList, only including the values of fruits that this fruit shop has.

""" totalCost = 0.0 for fruit, numPounds in orderList: costPerPound = self.getCostPerPound(fruit) if costPerPound != None: totalCost += numPounds \* costPerPound return totalCost

def getName(self): return self.name

The FruitShop class has some data, the name of the shop and the prices per pound of some fruit, and it provides functions, or methods, on this data. What advantage is there to wrapping this data in a class?

1. Encapsulating the data prevents it from being altered or used inappropriately, 2. The abstraction that objects provide make it easier to write general-purpose code.

## Using Objects

So how do we make an object and use it? Make sure you have the FruitShop implementation in shop.py. We then import the code from this file (making it accessible to other scripts) using import shop, since shop.py is the name of the file. Then, we can create FruitShop objects as follows:

import shop

shopName = 'the Berkeley Bowl'

fruitPrices = {'apples': 1.00, 'oranges': 1.50, 'pears': 1.75} berkeleyShop = shop.FruitShop(shopName, fruitPrices) applePrice = berkeleyShop.getCostPerPound('apples') print(applePrice)

print('Apples cost $%.2f at %s.' % (applePrice, shopName))

otherName = 'the Stanford Mall'

otherFruitPrices = {'kiwis':6.00, 'apples': 4.50, 'peaches': 8.75} otherFruitShop = shop.FruitShop(otherName, otherFruitPrices) otherPrice = otherFruitShop.getCostPerPound('apples') print(otherPrice)

print('Apples cost $%.2f at %s.' % (otherPrice, otherName)) print("My, that's expensive!")

This code is in shopTest.py; you can run it like this:

[cs188-ta@nova ~]$ python shopTest.py

Welcome to the Berkeley Bowl fruit shop

1.0

Apples cost $1.00 at the Berkeley Bowl.

Welcome to the Stanford Mall fruit shop

4.5

Apples cost $4.50 at the Stanford Mall. My, that's expensive!

So what just happended? The import shop statement told Python to load all of the functions and classes in shop.py. The line berkeleyShop = shop.FruitShop(shopName, fruitPrices) constructs an *instance* of the FruitShop class defined in *shop.py*, by calling the \_\_init\_\_ function in that class. Note that we only passed two arguments in, while \_\_init\_\_ seems to take three arguments: (self, name, fruitPrices). The reason for this is that all methods in a class have self as the first argument. The self variable's value is automatically set to the object itself; when calling a method, you only supply the remaining arguments. The self variable contains all the data (name and fruitPrices) for the current specific instance (similar to this in Java). The print statements use the substitution operator (described in the Python docs if you're curious).

## Static vs Instance Variables

The following example illustrates how to use static and instance variables in Python.

Create the person\_class.py containing the following code:

class Person:

population = 0 def \_\_init\_\_(self, myAge): self.age = myAge Person.population += 1 def get\_population(self): return Person.population def get\_age(self): return self.age

We first compile the script:

[cs188-ta@nova ~]$ python person\_class.py

Now use the class as follows:

>>> import person\_class

>>> p1 = person\_class.Person(12)

>>> p1.get\_population()

1

>>> p2 = person\_class.Person(63)

>>> p1.get\_population()

2

>>> p2.get\_population()

2

>>> p1.get\_age()

12

>>> p2.get\_age()

63

In the code above, age is an instance variable and population is a static variable. population is shared by all instances of the Person class whereas each instance has its own age variable.

## More Python Tips and Tricks

This tutorial has briefly touched on some major aspects of Python that will be relevant to the course. Here are some more useful tidbits:

* Use range to generate a sequence of integers, useful for generating traditional indexed for loops:
* for index in range(3):
* print(lst[index])
* After importing a file, if you edit a source file, the changes will not be immediately propagated in the interpreter. For this, use the reload command:

>>> reload(shop)

# Python Object 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 you define a new class, Python 3 it implicitly uses object as the parent class. So the following two definitions are equivalent: class Dog(object): pass

# In Python 3, this is the same as:

class Dog: pass

**Note:** In Python 2.x there’s a distinction between *new-style* and *old-style* classes. I won’t go into detail here, but you’ll generally want to specify object as the parent class to ensure you’re definint a new-style class if you’re writing Python 2 OOP code.

## Dog Park Example

Let’s pretend that we’re at a dog park. There are multiple Dog objects engaging in Dog behaviors, each with different attributes. In regular-speak that means some dogs are running, while some are stretching and some are just watching other dogs. Furthermore, each dog has been named by its owner and, since each dog is living and breathing, each ages.

What’s another way to differentiate one dog from another? How about the dog’s breed:

>>> class Dog:

... def \_\_init\_\_(self, breed): ... self.breed = breed

...

>>> spencer = Dog("German Shepard")

>>> spencer.breed

'German Shepard'

>>> sara = Dog("Boston Terrier")

>>> sara.breed

'Boston Terrier'

Each breed of dog has slightly different behaviors. To take these into account, let’s create separate classes for each breed. These are child classes of the parent Dog class.

**Extending the Functionality of a Parent Class** Create a new file called *dog\_inheritance.py*:

# Parent class class Dog:

# 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)

# Child class (inherits from Dog class) class RussellTerrier(Dog): def run(self, speed): return "{} runs {}".format(self.name, speed)

# Child class (inherits from Dog class) class Bulldog(Dog): def run(self, speed): return "{} runs {}".format(self.name, speed)

# Child classes inherit attributes and # behaviors from the parent class jim = Bulldog("Jim", 12) print(jim.description())

# Child classes have specific attributes

# and behaviors as well print(jim.run("slowly"))

Read the comments aloud as you work through this program to help you understand what’s happening, then before you run the program, see if you can predict the expected output.

You should see:

Jim is 12 years old

Jim runs slowly

We haven’t added any special attributes or methods to differentiate a RussellTerrier from a Bulldog, but since they’re now two different classes, we could for instance give them different class attributes defining their respective speeds.

## Parent vs. Child Classes

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

Save this as *dog\_isinstance.py*:

# Parent class class Dog:

# 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)

# Child class (inherits from Dog() class) class RussellTerrier(Dog): def run(self, speed): return "{} runs {}".format(self.name, speed)

# Child class (inherits from Dog() class) class Bulldog(Dog): def run(self, speed): return "{} runs {}".format(self.name, speed)

# Child classes inherit attributes and # behaviors from the parent class jim = Bulldog("Jim", 12) print(jim.description())

# Child classes have specific attributes

# and behaviors as well print(jim.run("slowly"))

# Is jim an instance of Dog()? print(isinstance(jim, Dog))

# Is julie an instance of Dog()? julie = Dog("Julie", 100) print(isinstance(julie, Dog))

# Is johnny walker an instance of Bulldog() johnnywalker = RussellTerrier("Johnny Walker", 4) print(isinstance(johnnywalker, Bulldog))

# Is julie and instance of jim? print(isinstance(julie, jim))

Output:

('Jim', 12)

Jim runs slowly

True

True

False

Traceback (most recent call last):

File "dog\_isinstance.py", line 50, in <module>

print(isinstance(julie, jim))

TypeError: isinstance() arg 2 must be a class, type, or tuple of classes and types

Make sense? Both jim and julie are instances of the Dog() class, while johnnywalker is not an instance of the Bulldog() class. Then as a sanity check, we tested if julie is an instance of jim, which is impossible since jim is an instance of a class rather than a class itself—hence the reason for the TypeError.

## Overriding the Functionality of a Parent Class

Remember that child classes can also override attributes and behaviors from the parent class. For examples:

>>> class Dog:

... species = 'mammal'

...

>>> class SomeBreed(Dog): ... pass

...

>>> class SomeOtherBreed(Dog): ... species = 'reptile'

...

>>> frank = SomeBreed()

>>> frank.species

'mammal'

>>> beans = SomeOtherBreed()

>>> beans.species

'reptile'

The SomeBreed() class inherits the species from the parent class, while the SomeOtherBreed() class overrides the species, setting it to reptile.

## LAB Exercises

1. Write a insertionSort function in Python using list comprehensions.

**Source Code:**

def insertion\_sort(nums):

for i in range(1, len(nums)):

item\_to\_insert = nums[i]

j = i - 1

while j >= 0 and nums[j] > item\_to\_insert:

nums[j + 1] = nums[j]

j -= 1

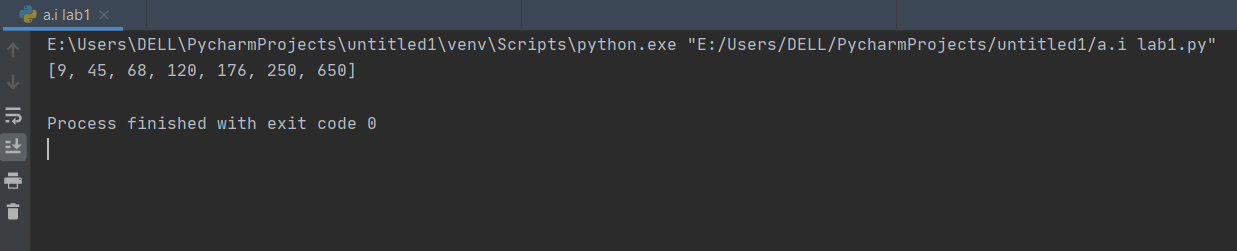
nums[j + 1] = item\_to\_insert

random\_list\_of\_nums = [9,650,120, 45, 68, 250, 176]

insertion\_sort(random\_list\_of\_nums)

print(random\_list\_of\_nums)

**Output:**



1. Write a mergeSort function in Python using list comprehensions.

**Source Code:**

cdef merge(left\_list, right\_list):

sorted\_list = []

left\_list\_index = right\_list\_index = 0

left\_list\_length, right\_list\_length = len(left\_list), len(right\_list)

for \_ in range(left\_list\_length + right\_list\_length):

if left\_list\_index < left\_list\_length and right\_list\_index < right\_list\_length:

if left\_list[left\_list\_index] <= right\_list[right\_list\_index]:

sorted\_list.append(left\_list[left\_list\_index])

left\_list\_index += 1

else:

sorted\_list.append(right\_list[right\_list\_index])

right\_list\_index += 1

elif left\_list\_index == left\_list\_length:

sorted\_list.append(right\_list[right\_list\_index])

right\_list\_index += 1

elif right\_list\_index == right\_list\_length:

sorted\_list.append(left\_list[left\_list\_index])

left\_list\_index += 1

return sorted\_list

def merge\_sort(nums):

if len(nums) <= 1:

return nums

mid = len(nums) // 2

left\_list = merge\_sort(nums[:mid])

right\_list = merge\_sort(nums[mid:])

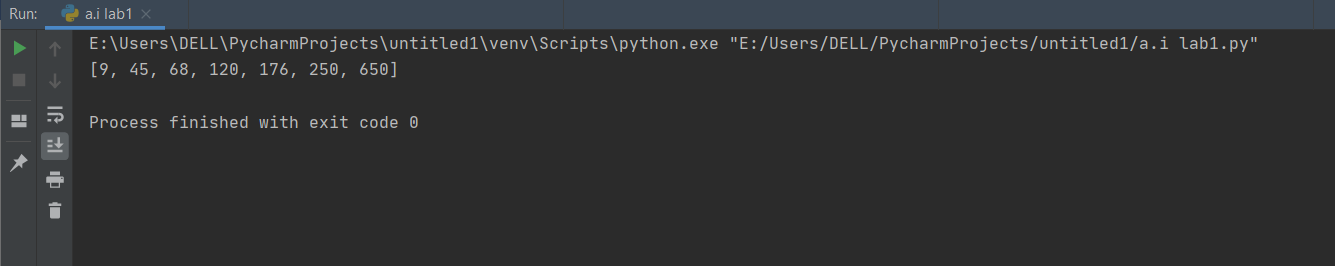
return merge(left\_list, right\_list)

random\_list\_of\_nums = [9, 650, 120, 45, 68, 250, 176]

random\_list\_of\_nums = merge\_sort(random\_list\_of\_nums)

print(random\_list\_of\_nums)

**Output:**



1. Write a quickSort function in Python using list comprehensions. Use the first element as the pivot.

**Source Code:**

def partition\_Function(nums, l, h):

# l means low & h means high

pivot = nums[(l + h) // 2]

i = l - 1

j = h + 1

while True:

i += 1

while nums[i] < pivot:

i += 1

j -= 1

while nums[j] > pivot:

j -= 1

if i >= j:

return j

nums[i], nums[j] = nums[j], nums[i]

def quick\_sort(nums):

def quick\_sort\_helper(items, l, h):

if l < h:

split\_index = partition\_Function(items, l, h)

quick\_sort\_helper(items, l, split\_index)

quick\_sort\_helper(items, split\_index + 1, h)

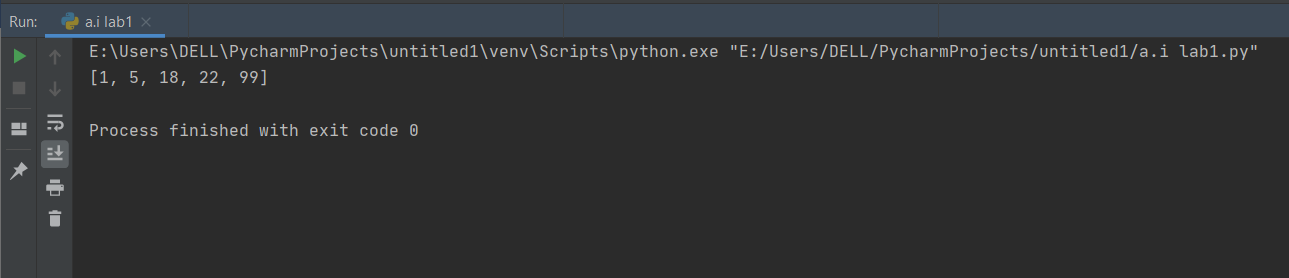
quick\_sort\_helper(nums, 0, len(nums) - 1)

random\_list\_of\_nums = [22, 5, 1, 18, 99]

quick\_sort(random\_list\_of\_nums)

print(random\_list\_of\_nums)

**Output:**



1. For the following Dog class, do the following exercise

class Dog:

# Class Attribute species = 'mammal'

# Initializer / Instance Attributes def \_\_init\_\_(self, name, age): self.name = name self.age = age

# Instantiate the Dog object philo = Dog("Philo", 5) mikey = Dog("Mikey", 6)

# Access the instance attributes print("{} is {} and {} is {}.".format(

philo.name, philo.age, mikey.name, mikey.age))

# Is Philo a mammal? if philo.species == "mammal": print("{0} is a {1}!".format(philo.name, philo.species))

Using the same Dog class, instantiate three new dogs, each with a different age. Then write a function called, get\_biggest\_number(), that takes any number of ages (\*args) and returns the oldest one. Then output the age of the oldest dog like so: x

**Source Code:**

class Dog:

species = 'mammal'

def \_\_init\_\_(self,name,age):

self.age = age

self.name = name

philo = Dog("Philo",5)

mikey = Dog("Mikey",10)

tommy= Dog("Tommy",15)

print("{} is {} and {} is {}.".format( philo.name, philo.age, mikey.name, mikey.age))

if philo.species == "mammal":

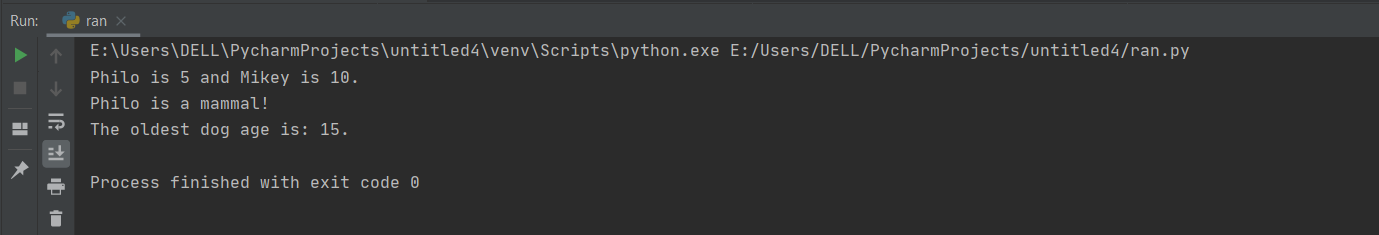
print("{0} is a {1}!".format(philo.name, philo.species))

def get\_biggest\_number(\*args):

return max(args)

print("The oldest dog age is: {}.".format(get\_biggest\_number(philo.age,mikey.age,tommy.age)))

**Output:**



Create a Pets class that holds instances of dogs; this class is completely separate from the Dog class. In other words, the Dog class does not inherit from the Pets class. Then assign three dog instances to an instance of the Pets class. Start with the following code below. Save the file as *pets\_class.py*. Your output should look like this:

I have 3 dogs. Tom is 6.

Fletcher is 7.

Larry is 9.

And they're all mammals, of course.

**Source Code:**

# Parent class

class Pets:

dogs = []

def \_\_init\_\_(self, dogs):

self.dogs = dogs

# Parent class

class Dog:

# Class attribute

species = 'mammal'

# Initializer / Instance attributes

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

# Instance method

def description(self):

return self.name, self.age

# Instance method

def speak(self, sound):

return "%s says %s" % (self.name, sound)

# Instance method

def eat(self):

self.is\_hungry = False

class RussellTerrier(Dog):

def run(self, speed):

return "%s runs %s" % (self.name, speed)

class Bulldog(Dog):

def run(self, speed):

return "%s runs %s" % (self.name, speed)

my\_dogs = [

Bulldog("Tom", 6),

RussellTerrier("Fletcher", 7),

Dog("Larry", 9)

]

my\_pets = Pets(my\_dogs)

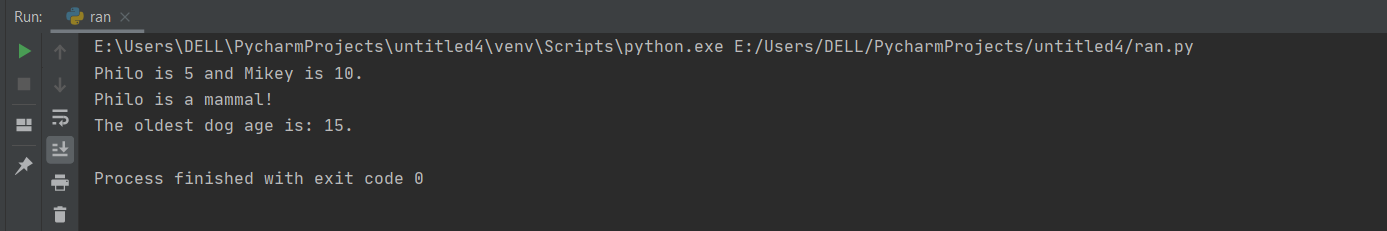
print("I have {} dogs.".format(len(my\_pets.dogs)))

for dog in my\_pets.dogs:

print("{} is {}.".format(dog.name, dog.age))

print("And they're all {}s, of course.".format(dog.species))

**Output:**



5.Predict the result of evaluating the following calls in the interpreter. Then try them out yourself!

>>> class Account(object):

... interest = 0.02

... def \_\_init\_\_(self, account\_holder):

... self.balance = 0

... self.holder = account\_holder

... def deposit(self, amount):

... self.balance = self.balance + amount ... print("Yes!")

>>> class CheckingAccount(Account):

... def \_\_init\_\_(self, account\_holder):

... Account.\_\_init\_\_(self, account\_holder)

... def deposit(self, amount):

... Account.deposit(self, amount) ...

print("Have a nice day!")

...

>>> a = Account("Billy")

>>> a.balance

\_\_\_\_\_\_

>>> c = CheckingAccount("Eric")

>>> c.balance

\_\_\_\_\_\_

>>> a.deposit(30)

\_\_\_\_\_\_

>>> c.deposit(30)

\_\_\_\_\_\_

>>> c.interest

\_\_\_\_\_\_

**Source Code:**

class Account(object):

interest = 0.02

def \_\_init\_\_(self, account\_holder):

self.balance = 0

self.holder = account\_holder

def deposit(self, amount):

self.balance = self.balance + amount

print("Yes!")

class CheckingAccount(Account):

def \_\_init\_\_(self, account\_holder):

Account.\_\_init\_\_(self, account\_holder)

def deposit(self, amount):

Account.deposit(self, amount)

print("Have a nice day!")

a = Account("Billy")

a.balance

c = CheckingAccount("Eric")

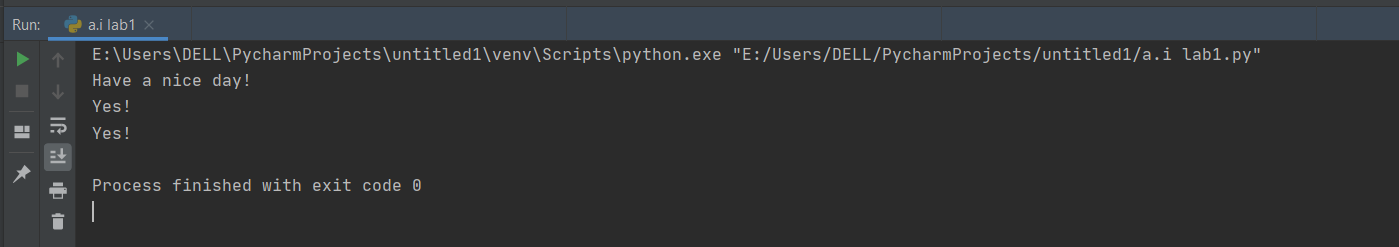
c.balance

a.deposit(30)

c.deposit(30)

c.interest

**Output:**



## 6. Suppose now that we wanted to define a class called DoubleTalker to represent people who always say things twice:

>>> steven = DoubleTalker("Steven")

>>> steven.say("hello")

"hello hello"

>>> steven.say("the sky is falling")

"the sky is falling the sky is falling"

Consider the following three definitions for DoubleTalker that inherit from the Person class:

class DoubleTalker(Person):

def \_\_init\_\_(self, name):

Person.\_\_init\_\_(self, name)

def say(self, stuff):

return Person.say(self, stuff) + " " + self.repeat()

class DoubleTalker(Person):

def \_\_init\_\_(self, name):

Person.\_\_init\_\_(self, name)

def say(self, stuff):

return stuff + " " + stuff

class DoubleTalker(Person):

def \_\_init\_\_(self, name):

Person.\_\_init\_\_(self, name)

def say(self, stuff):

return Person.say(self, stuff + " " + stuff)

Determine which of these definitions work as intended. Also determine for which of the methods the three versions would respond differently. (Don't forget about the repeat method!)

**Source Code:**

class Person(object):

def \_\_init\_\_(self, value):

self.value = value

def say(self, stuff):

self.value = stuff

return self.value

def ask(self, stuff):

return self.say("Would you please " + stuff)

def greet(self, value):

return self.say("Hello, my name is " + value)

def repeat(self):

return self.value

class DoubleTalker(Person):

def \_\_init\_\_(self, name):

Person.\_\_init\_\_(self, name)

def say(self, stuff):

return print(Person.say(self, stuff) + " " + self.repeat())

#print('DoubleTalker program!')

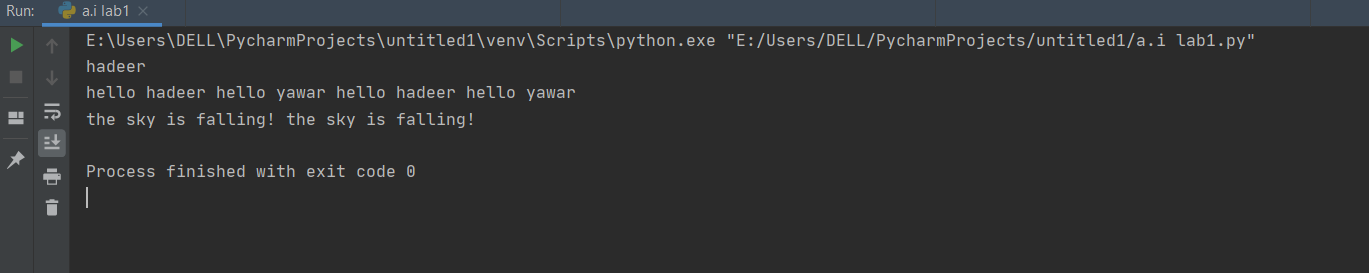
steven = DoubleTalker("hadeer")

print(steven.value)

steven.say("hello hadeer hello yawar")

steven.say('the sky is falling!')

**Output:**



7.Write a Python class named Circle constructed by a radius and two methods which will compute the area and the perimeter of a circle

**Source Code:**

import math

class cir():

def \_\_init\_\_(self, radius):

self.radius = radius

def area(self):

return math.pi \* (self.radius \*\* 2)

def perimeter(self):

return 2 \* math.pi \* self.radius

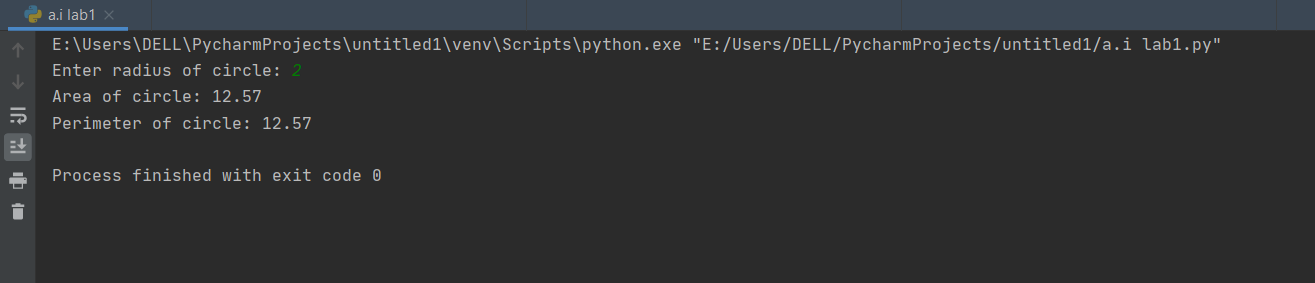
r = int(input("Enter radius of circle: "))

obj = cir(r)

print("Area of circle:", round(obj.area(), 2))

print("Perimeter of circle:", round(obj.perimeter(), 2))

**Output:**



8. Fashion is a famous article of clothing store in USA. The store has numerous areas for men, ladies, and youngsters for various sorts of pieces of clothing, for example, pants, suits, customary wear, embellishments, athletic equipment, etc. As of now, the store keeps up a manual register for recording the subtleties of the articles of clothing, for example, type, value, stock, orderQty, deals, etc. In any case, the proprietor of the store has chosen to mechanize the errands, for example, information section, deals estimation, stock updation, etc to spare time and exertion required in theundertakings.

You as a product engineer need to achieve the errand as follows:

1. Make a class named Garments.java.

2. Make factors ID, type, stock, value, deals, and orderQty with fitting information types and access specifiers to limit access from outside the class.

3. Make constructor to introduce the estimations of the factors as per the contentions determined by client at order line.

4. Make strategy addGarment() that adds the piece of clothing subtleties to an assortment.

5.Make strategy show Detail so to show the subtleties of the piece of clothing from the assortment.

6. Make a strategy setOrder() to acknowledge the orderQty for the present piece of clothing and figure the business esteem by increasing the cost with the ordarOlv. Show the payable add up to the client.

**Source Code:**

class Garments:

factors = 100

types = "cotton"

value = 200

stock = 6000

orderQty = 3500

def showdetails(self):

print(self.factors, self.types, self.value, self.stock, self.orderQty)

print("Garments Details:")

print("Garments ids:", self.factors)

print("Garments types:", self.types)

print("Garments stock:", self.stock)

print("Garments quantity:", self.orderQty)

g = Garments()

g.showdetails()

def add\_garments():

stock = 60

inwarehouse = 30

if (stock > 50):

print("It has been added.")

elif (inwarehouse > 50):

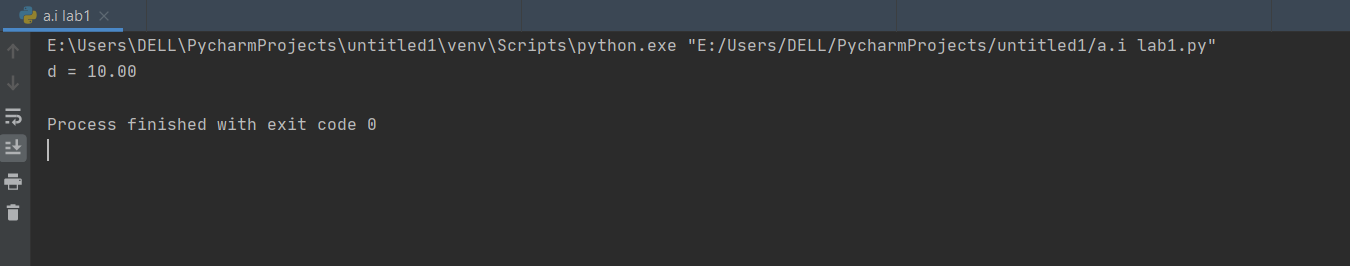
print("We have some more quality of garments")

else:

print("get some new materials ", stock + inwarehouse)

add\_garments()

**Output:**



9- Design a class named Account that contains:

-A private int data field named id for the account (default 0).

- A private double data field named balance for the account (default 0).

- A private double data field named annualInterestRate that stores the current interest rate (default 0). Assume all accounts have the same interest rate.

- A private Date data field named dateCreated that stores the date when the account was

created.

- A no-arg constructor that creates a default account.

- A constructor that creates an account with the specified id and initial balance.

- The accessor and mutator methods for id, balance, and annualInterestRate.

- The accessor method for dateCreated.

- A method named getMonthlyInterestRate() that returns the monthly interest rate.

- A method named getMonthlyInterest() that returns the monthly interest.

- A method named withdraws that withdraws a specified amount from the account.

- A method named deposit that deposits a specified amount to the account.

(Hint: Monthly interest is balance \* monthlyInterestRate.

monthlyInterestRate is annualInterestRate / 12.

Note that annualInterestRate is a percentage. You need to divide it by 100.

Write a test program that creates an Account object with an account ID of 1122, a balance of

$20,000, and an annual interest rate of 4.5%. Use the withdraw method to withdraw $2,500, use

the deposit method to deposit $3,000, and print the balance, the monthly interest, and the date

when this account was created.

**Source Code:**

import datetime

class Account(object):

id = 0;

balance = 0.0;

annualInterestRate = 0.0;

def \_\_init\_\_(self,id,balance):

self.id = id

self.balance = balance

def getId() :

return this.id

def getBalance(self) :

return self.balance

def getAnnualInterestRate(self) :

return annualInterestRate

def getDateCreated(self) :

return str(datetime.datetime.now())

def setId(self,id):

self.\_id = id

def setBalance(self,balance):

self.\_balance = balance

def setAnnualInterestRate(self,annualInterestRate):

self.annualInterestRate = annualInterestRate

def getMonthlyInterestRate(self):

return (self.annualInterestRate / 100) / 12

def getMonthlyInterest(self):

return self.balance \* self.getMonthlyInterestRate()

def withdraw(self,amount):

self.balance -= amount

def deposit(self,amount):

self.balance += amount

class BankAccount:

account =Account(1122, 20000)

account.setAnnualInterestRate(4.5)

account.withdraw(2500.0)

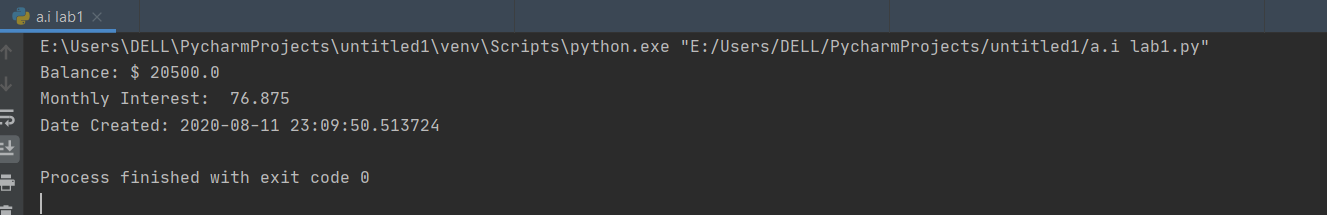
account.deposit(3000.0)

print("Balance: $" ,account.getBalance())

print("Monthly Interest: " ,account.getMonthlyInterest())

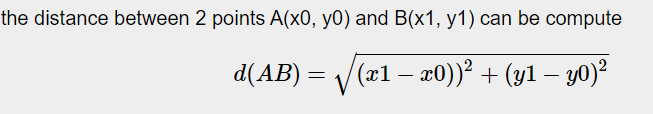
print("Date Created: " + account.getDateCreated())

Output:



10. Write the definition of a Point class. Objects from this class should have a

* a method show to display the coordinates of the point
* a method move to change these coordinates.
* a method dist that computes the distance between 2 points.



Source Code:

import math

class Point:

def dist(x0 , y0 , x1 , y1):

return math.sqrt(math.pow(x1 - x0, 2) + math.pow(y1 - y0, 2) \* 1.0)

print("d = %.2f"%dist(0, 0, 6, 8))

Output:

