Master of Engineering in Internetworking

INWK 6312

Programming for Internetworking Applications

Lab 5

Classes (Objects, Methods and Inheritance)

TABLE OF CONTENTS

CHAPTER 1 - INTRODUCTION

This lab is to introduces you to the Object-Oriented programming in python. You have learned structural/functional programming in the last 2 weeks. We will now look at classes and objects, functions and methods and the advantages of using them.

CHAPTER 2 - OBJECTIVES OF THE LAB

At the end of this lab you will have learnt the following:

- Classes and objects
- Functional programming
- Object oriented programming
- Classes and Functions
- Classes and Methods
- Difference between classes and methods
- Built-in methods
- Constructors, Inheritance and Encapsulation

CHAPTER 3 - CLASSES AND OBJECTS

Getting started with Classes and Objects

class: A user-defined type. A class definition creates a new class object.

class object: An object that contains information about a user-defined type. The class object can be used to create instances of the type.

instance: An object that belongs to a class.

attribute: One of the named values associated with an object.

embedded (object): An object that is stored as an attribute of another object.

shallow copy: To copy the contents of an object, including any references to embedded objects; implemented by the copy function in the copy module.

deep copy: To copy the contents of an object as well as any embedded objects, and any objects embedded in them, and so on; implemented by the deepcopy function in the copy module.

object diagram: A diagram that shows objects, their attributes, and the values of the attributes.

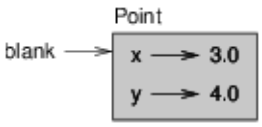
Study the following code snippet and type in a Python interpreter:

```
class Point(object): #creates a new class

    """Represents a point in 2-D space."""
>>> print(Point)
<class '__main__.Point'>

>>> blank = Point()  # creating an instance of a "Point" object
>>> print(blank)
<__main__.Point instance at 0xb7e9d3ac>
>>> blank.x = 3.0
>>> blank.y = 4.0
```

Object Diagram



Task 1

Create a new class called Point. This class will have a "x" and "y" attribute.

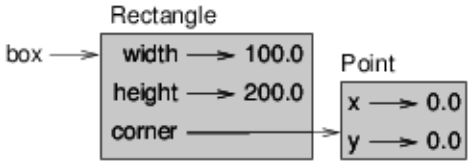
write a function called distance_between_points *that takes two Points as arguments and returns the distance between them.*

Test you function by instantiating two instances and assigning them x and y attributes.

Task 2

Create a new class called Rectangle, this class will have width, height and corner attributes. ***The corner attribute is an instance of the Point class created in task1.***

The object diagram of an instance of the class is below:



A. *Write a function called "find_center" that takes a Rectangle has an argument and returns a Point that contains the coordinates of the center of the Rectangle. (Assuming the corner of the rectangle is on the origin)*

B. *Write a function named move_rectangle that takes a Rectangle and two numbers named dx and dy. It should change the location of the rectangle by adding dx to the x coordinate of corner and adding dy to the y coordinate of corner.*

CHAPTER 4 - CLASSES AND FUNCTIONS

Important concepts

prototype and patch: A development plan that involves writing a rough draft of a program, testing, and correcting errors as they are found.

planned development: A development plan that involves high-level insight into the problem and more planning than incremental development or prototype development.

pure function: A function that does not modify any of the objects it receives as arguments. Most pure functions are fruitful.

modifier: A function that changes one or more of the objects it receives as arguments. Most modifiers are fruitless.

functional programming style: A style of program design in which most of functions are pure.

invariant: A condition that should always be true during the execution of a program.

CHAPTER 5 - CLASSES AND METHORDS

Important concepts

object-oriented language: A language that provides features, such as user-defined classes and method syntax, that facilitate object-oriented programming.

object-oriented programming: A style of programming in which data and the operations that manipulate it are organized into classes and methods.

method: A function that is defined inside a class definition and is invoked on instances of that class.

subject: The object a method is invoked on.

operator overloading: Changing the behavior of an operator like + so it works with a user-defined type.

type-based dispatch: A programming pattern that checks the type of an operand and invokes different functions for different types.

polymorphic: Pertaining to a function that can work with more than one type.

information hiding: The principle that the interface provided by an object should not depend on its implementation, the representation of its attributes

REFERENCE: BASIC OBJECT CUSTOMIZATIONS

This list is not exhaustive, visit the python documentation for more:

object.**__new__**(*cls*[, ...])

> Called to create a new instance of class *cls*. __new__() is a static method (special-cased so you need not declare it as such) that takes the class of which an instance was requested as its first argument. The remaining arguments are those passed to the object constructor expression (the call to the class). The return value of __new__() should be the new object instance (usually an instance of *cls*).

object.**__init__**(*self*[, ...])

> Called after the instance has been created (by __new__()), but before it is returned to the caller. The arguments are those passed to the class constructor expression. If a base class has an __init__() method, the derived class's __init__() method, if any, must explicitly call it to ensure proper initialization of the base class part of the instance; for example: BaseClass.__init__(self,[args...]).
>
> Because __new__() and __init__() work together in constructing objects (__new__() to create it, and __init__() to customise it), no non-None value may be returned by __init__(); doing so will cause a TypeError to be raised at runtime.

object.**__del__**(*self*)

> Called when the instance is about to be destroyed. This is also called a destructor. If a base class has a __del__() method, the derived class's __del__() method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance. Note that it is possible (though not recommended!) for the __del__() method to postpone destruction of the instance by creating a new reference to it. It may then be called at a later time when this new reference is deleted. It is not guaranteed that __del__() methods are called for objects that still exist when the interpreter exits.

object.**__repr__**(*self*)

> Called by the repr() built-in function and by string conversions (reverse quotes) to compute the "official" string representation of an object. If at all possible, this should look like a valid Python expression that could be used to recreate an object with the same value (given an appropriate environment). If this is not possible, a string of the form <...some useful description...> should be

returned. The return value must be a string object. If a class defines __repr__() but not __str__(), then __repr__() is also used when an "informal" string representation of instances of that class is required.

This is typically used for debugging, so it is important that the representation is information-rich and unambiguous.

object.**__str__**(*self*)

Called by the str() built-in function and by the print statement to compute the "informal" string representation of an object. This differs from __repr__() in that it does not have to be a valid Python expression: a more convenient or concise representation may be used instead. The return value must be a string object.

object.**__lt__**(*self*, *other*), object.**__le__**(*self*, *other*), object.**__eq__**(*self*, *other*), object.**__ne__**(*self*, *other*), object.**__gt__**(*self*, *other*), object.**__ge__**(*self*, *other*)

These are the so-called "rich comparison" methods, and are called for comparison operators in preference to __cmp__() below. The correspondence between operator symbols and method names is as follows:

x<y calls x.__lt__(y),

x<=y calls x.__le__(y),

x==y calls x.__eq__(y),

x!=y and x<>y call x.__ne__(y),

x>y calls x.__gt__(y), and

x>=y calls x.__ge__(y).

Task 3

Write a constructor(__init__ method) for the Point class that takes x and y as optional parameters and assigns them to the corresponding attributes.

A. *Write a __str__ method for the Point class. Test by creating a Point object and use the print function to print an instance of the class.*

B. *Write an __add__ method for the Point class.*

Task 4

Create a Time class that has instant attributes: hour, minutes and seconds.

A. Write a __str__ method for the Time class to print an output in the format: 02:34:10
Hint: Try to use the "format" method of String

B. *Write a boolean **function** called is_after that takes two **Time objects**, t1 and t2, and returns True if t1 follows t2 chronologically and False otherwise. Challenge: don't use an if statement*

Task 5

This exercise is a cautionary tale about one of the most common, and difficult to find, errors in Python. Write a definition for a class named Kangaroo with the following methods:

1. *An __init__ method that initializes an attribute named pouch_contents to an empty list.*
2. *A method named put_in_pouch that takes an object of any type and adds it to pouch_contents.*
3. *A __str__ method that returns a string representation of the Kangaroo object and the contents of the pouch.*

Test your code by creating two Kangaroo objects, assigning them to variables named kanga and roo, and then adding roo to the contents of kanga's pouch.

CHAPTER 6 - INHERITANCE

Inheritance is a common practice (in object programming) of passing attributes and methods from the superclass (defined and existing) to a newly created class, called the subclass.

In other words, inheritance is a way of building a new class, not from scratch, but **by using an already defined repertoire of traits**. The new class inherits (and this is the key) **all the already existing equipment**, but **is able to add some new** ones if needed. Thanks to that, it's possible to build more specialized (more concrete) classes using some sets of predefined general rules and behaviors.

The most important factor of the process is the relation between the superclass and all of its subclasses (note: if B is a subclass of A and C is a subclass of B, this also means than C is a subclass of A, as the relationship is fully transitive).

```
class Vehicle:
      pass
class LandVehicle(Vehicle):
      pass
class TrackedVehicle(LandVehicle):
      pass
```

A very simple example of two-level inheritance is presented here above

All the presented classes are empty, as we're going to show how the mutual relations between the super- and subclasses work.

We can say that:

The Vehicle class is the superclass for both the LandVehicle and TrackedVehicle classes;

The LandVehicle class is a subclass of Vehicle and a superclass of TrackedVehicle at the same time;

The TrackedVehicle class is a subclass of both the Vehicle and LandVehicle classes.

The above knowledge comes from reading the code (in other words, we know it because we can see it). Does Python know the same? Is it possible to ask it about it? Yes, it is.

Python offers a function which is able to identify a relationship between two classes, and although its diagnosis isn't complex, it can check if a particular class is a subclass of any other class.

>> issubclass(class1, class2)

The function returns True if class1 is a subclass of class2, and False otherwise.

Task 6

Read the following code and fill in the table below with either True or False:

```
class Vehicle:
      pass
class LandVehicle(Vehicle):
      pass
class TrackedVehicle(LandVehicle):
      pass
for cl1 in [Vehicle, LandVehicle, TrackedVehicle]:
      for cl2 in [Vehicle, LandVehicle, TrackedVehicle]:
            print(issubclass(cl1,cl2),end='\t')
      print()
```

↓ is a subclass of →	Vehicle	LandVehicle	TrackedVehicle
Vehicle			
LandVehicle			
TrackedVehicle			

As you already know, an object is an incarnation of a class. This means that the object is like a cake baked using a recipe which is included inside the class.

Similarly, it can be crucial if the object does have (or doesn't have) certain characteristics. In other words, whether it is an object of a certain class or not.

Such a fact could be detected by the function named **isinstance()**

The function returns True if the object is an instance of the class, or False otherwise.

Being an instance of a class means that the object (the cake) has been prepared using a recipe contained in either the class or one of its superclasses.

Don't forget: if a subclass contains at least the same equipment as any of its superclasses, it means that objects of the subclass can do the same as objects derived from the superclass, ergo, it's an instance of its home class and any of its superclasses.

Task 7:

Read the following code and fill in the table below with either True or False:

```
class Vehicle:
      pass
class LandVehicle(Vehicle):
      pass
class TrackedVehicle(LandVehicle):
      pass

vehicle = Vehicle()
landvehicle = LandVehicle()
trackedvehicle = TrackedVehicle()

for ob in [vehicle, landvehicle, trackedvehicle]:
      for cl in [Vehicle, LandVehicle, TrackedVehicle]:
            print(isinstance(ob,cl),end='\t')
      print()
```

↓ is instance of →	Vehicle	LandVehicle	TrackedVehicle
Vehicle			
LandVehicle			
TrackedVehicle			

CHAPTER 7 - HOW PYTHON FINDS PROPERTIES AND METHODS

Now we're going to look at how Python deals with inheriting methods.

Take a look at the example →

```
class Super:
      def __init__(self,name):
            self.name = name
      def __str__(self):
            return "My name is " + self.name + "."
class Sub(Super):
      def __init__(self,name):
            Super.__init__(self,name)
object = Sub("Bob")
print(object)
```

There is a class named Super, which defines its own constructor used to assign the object's property, named name. The class defines the __str__() method, too, which makes the class able to present its identity in clear text form.

The class is next used as a base to create a subclass named Sub. The Sub class defines its own constructor, which invokes the one from the superclass. Note how we've done it:

```
Super.__init__(self,name)
```

We've explicitly named the superclass, and pointed to the method to invoke __init__(), providing all needed arguments.

We've instantiated one object of class Sub and printed it.

The code outputs:

My name is Bob.

Note: As there is no __str__() method within the Sub class, the printed string is to be produced within the Super class. This means that the __str__() method has been inherited by the Sub class.

****

```
class Super:
      def __init__(self,name):
            self.name = name
      def __str__(self):
            return "My name is " + self.name + "."
class Sub(Super):
      def __init__(self,name):
            super().__init__(name)
object = Sub("Bob")
print(object)
```

We've modified the code to show you another method of accessing any entity defined inside the superclass. In the last example, we explicitly named the superclass. In this example, we make use of the super() function, which accesses the superclass without needing to know its name →

The super() function creates a context in which you don't have (moreover, you mustn't) pass the self argument to the method being invoked – this is why it's possible to activate the superclass constructor using only one argument.

Note: you can use this mechanism not only to invoke the superclass constructor, but also to get access to any of the resources available inside the superclass.

HOMEWORK

Q1

Write an add method for Points that works with either a Point object or a tuple:

- *If the second operand is a Point, the method should return a new Point whose x coordinate is the sum of the x coordinates of the operands, and likewise for the y coordinates.*
- *If the second operand is a tuple, the method should add the first element of the tuple to the x coordinate and the second element to the y coordinate, and return a new Point with the result.*

Q2

Write Python class representing an IPv4 address.

The IP address class should be instantiated by a four-item list (where the first item in the list is the highest octet in the address) and a mask (should be represented by an integer of value 0 to 32)

the class should have at least the following methods:

- method getNetwork() returns a list representing the network portion of the IP address
- method getMask() return a list representing the mask as four octets (i.e. 255.255....)
- method getAddress() returns a list representing the address

Also implement __str__ method to print the IP address

Example:

```
ipv4 = IP4Addres([10,0,1,7], 24)
net = ipv4.getNetwork()                 # result : [10,0,1,0]
mask = ipv4.getMask()                   # result : [255,255,255,0]
```

[Bonus: Implement a wildcard mask]

Q3

Use the previous class to create functions for a subnet calculator.

The calculator takes a IP address and its mask and produces the following.

- The Network Address, Broadcast Address
- The first host, last host, total number of hosts