COMP2123

Self-learning Report

Objected-Oriented Programming and Python

Wong Ngai Sum 3035380875 - 18/12/2018

Objectives

At the end of this self learning lab, you should be able to:

- understand better the object-oriented approach in programming.
- design programs to solve real world problems based on object-oriented principles.

Objects in Python

Python is an object-oriented language. It supports many different kinds of data:

100 1.23 “Hi” [1, 2, 3, 4, 5] {“Name”: ”Peter”}

Everything in Python is an object and every object has:

- a **type**
- an internal **data representation** (primitive or composite)
- a set of procedures for **interaction** with the object

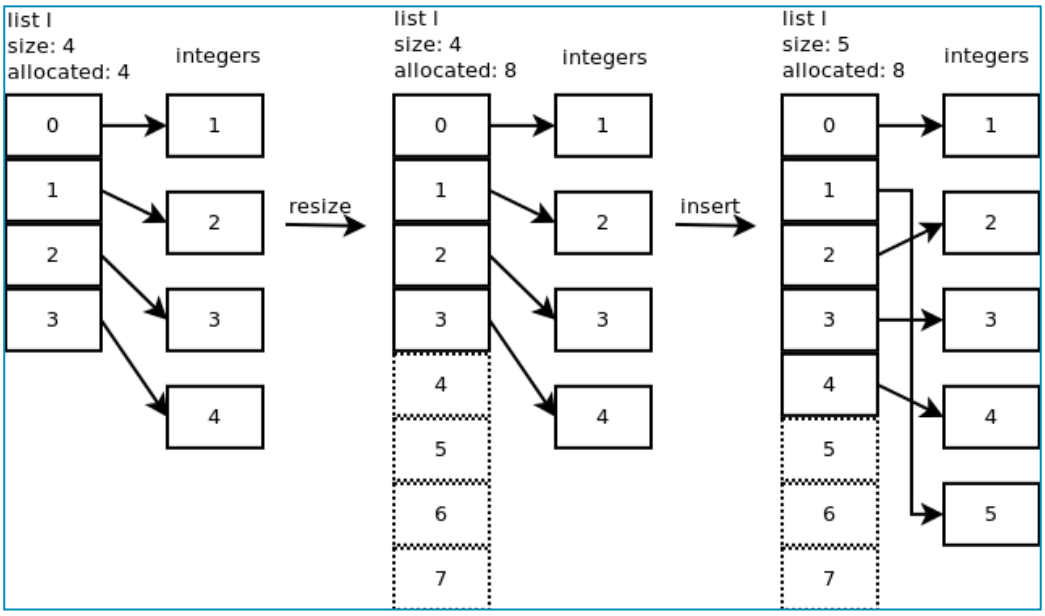
An object is an **instance** of a type, such as 123 is an instance of int, “Hi” is an instance of a string. We can **create** new objects of some type, **manipulate**, and **destroy** them.

Note:

A class is a **blueprint** that you use to create objects. “Object” and “Instance” are the same thing, but the word “Instance” the relationship of an object to its class. **Creating** a class involves defining the class name and its attributes. **Using** a class involves creating new **instances** of objects and doing operations on the instances.

Example: L = [1, 2, 3, 4, 5] has **type** list

- How is it **represented internally**? Linked list of cells.



- How to **manipulate** lists?
  - L[i], L[i:j], +
  - len(), min(), max(), del(L[i])
  - append(), clear(), copy(), count(), extend(), index(), insert(), pop(), …

Internal representation of a type should be private. If you manipulate it directly without built-in methods, correct behaviour may be compromised.

Define your own type

You can use the class keyword to define a new type:

The word **object** indicates Rectangle is a Python object and **inherits** all its attributes.

```
class Rectangle(object):
        # attributes
```

- Rectangle is a subclass of object
- object is a superclass of Rectangle

First, we should define **the way to create** an instance. We can use a method (function associated with an object) called **__init__** to initialise some attributes (similar to constructor in C++). Note that __ is double underscore and "self" refers to an instance of the class.

```
class Rectangle(object):
        def __init__(self, width, height):
                self.width = width
                self.height = height
```

To actually create an instance of the class, we can type

```
r1 = Rectangle(10, 5)
r2 = Rectangle(100, 1)
print(r1.width)
```

Data attributes of an instance are called **instance variables**.

You don't need to provide argument for "self" as Python does it automatically.

Define a method for the class

```
class Rectangle(object):
        def __init__(self, width, height):
                self.width = width
                self.height = height
        def area(self):
                return (self.width * self.height)
```

You need to use "self" to refer to **any instance** and **dot notation to access data**.

We can use a method in conventional way,

```
r1 = Rectangle(10, 5)
r2 = Rectangle(100, 1)
print(r1.area())
```

or by including an object to represent "self" to call the method.

```
r1 = Rectangle(10, 5)
r2 = Rectangle(100, 1)
print(Rectangle.area(r1))
```

Define “print” method for the class

If we define **__str__ method** for a class, Python calls the __str__ method when used with print() on the instances.

```
class Rectangle(object):
        def __init__(self, width, height):
                self.width = width
                self.height = height
        def area(self):
                return (self.width * self.height)
        def __str__(self):
                return “Rectangle width=” + str(self.width) + “ height=” + str(self.height)
```

Operators Overriding

Like __str__, you can override many other operators with your class to perform addition, subtraction, comparison and so on.

object.__**lt**__(self, other)

object.__**le**__(self, other)

object.__**eq**__(self, other)

object.__**ne**__(self, other)

object.__**gt**__(self, other)

object.__**ge**__(self, other)

… and a lot more

What is Object-Oriented Programming (OOP) ?

OOP is one of the most effective approaches to writing software. In OOP you write classes that represent real-world things, and you create objects based on these classes. OOP is based on a small number of common-sense fundamentals.

Four key principles:

i. **Abstraction**

ii. **Encapsulation**

iii. **Inheritance**

iv. **Polymorphism**

Knowing the logic behind classes will train you to think logically so you can write programs that effectively address almost any program you encounter. Classes make life easier for you when you take on complex challenges. Your programs will make sense to others, allowing everyone to accomplish more.

i. Abstraction

Abstraction means **taking away unimportant details**, keeping only "interesting" characteristics (including operations). For example,

Bus Company: License, Driver, Vehicle Model, …

Customer: Location, Fare, …

Abstraction is not a new thing: **procedures and functions**. Procedure abstraction abstract actions by parameterisation and specification.

Parameterisation abstracts from the identity of data being used, allowing the procedure to be used in more situations.

Relevant: presence, number, types of parameters

Irrelevant: identity of parameters

Specification abstracts from the realisation of the action, allowing multiple implementations.

Relevant: what is done

Irrelevant: how it is done

Both the code and data associated is encapsulated in the type specification, there is **no need to expose the representation** to clients, just like header files in C++.

ii. Encapsulation

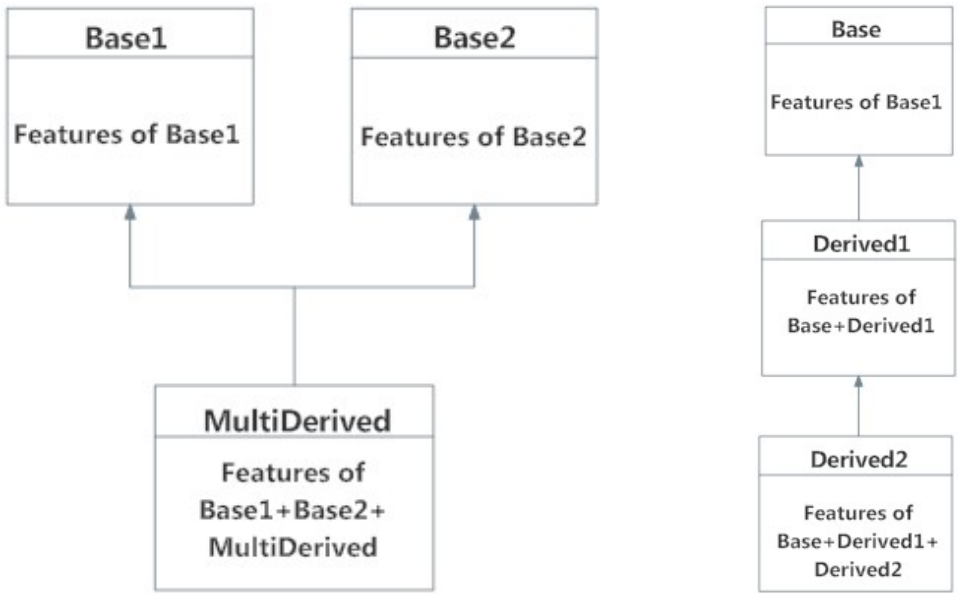
You can **restrict access** to methods and variables. For example,
You cannot access boot method by calling pc.__boot(), as encapsulation prevents it from accessing accidentally. Also, __name is private variable, it cannot be changed outside of a class. If you want to change its value, you should use a setter method.

Abstraction and encapsulation gives you more control over the degree of coupling in your code and allows changing its implementations without affecting other parts of the code.

```
class Computer(object):
        __name = ""
        def __init__(self):
                self.__boot()
                self.__name = "PC"
        def setName(self, name):
                self.__name = name
        def __boot(self):
                print("booting")
pc = Computer()
```

iii. Inheritance

You don't need to start from scratch when writing a class. If the class you're writing is a specialised version of another class you wrote, you can use **inheritance**. When one class inherits from another, it **takes on all the attributes and methods of the first class**. The original class is called the **parent class (superclass)**, and the new class is the **child class (subclass)**. The child class is free to define new attributes and methods of its own.



For example, there are lots of animals in a zoo, like lions, dogs, cats, wolfs, hippos, tigers and so on. They both have similarities and we can abstract out them and define the

inheritance tree relationships. A class **Animal** contains the **common state** (instance variables) and **behaviour** (methods), such as animal picture, hunger level, location, eat(), and makeNoise(). They can be overridden such that each animal can define its own way of behaviours if needed.

```
class Animal(object):
        def __init__(self, location, food, hunger, boundaries):
                self.location = location
                self.food = food
                self.hunger = hunger
                self.boundaries = boundaries
class  Hippo(Animal):
        def __init__(self, location, food, hunger, boundaries):
                super().__init__(location, food, hunger, boundaries)
        def makeNoise():
                print("Hippo making noise")
        def eat():
                print("Hippo eating")
```

__init__() method in subclass Hippo invokes that __init__() in Animal class, it means that same method in subclass gets preference over the same in superclass.

Multiple Inheritance

Python is not Java, it allows us to **derive a class from several class**, which is known as multiple inheritance. Its format is as follows:

```
class Parent1:
        #body
class Parent2:
        #body
class Child(Parent1, Parent2):
        #body
```

iv. Polymorphism

Polymorphism is the state of being in lots of shapes or forms. Python functions are polymorphic, they can be **applied to arguments of different types** and act differently based on types of arguments. For example,

```
def echo(i):
        return i
print(echo(1))
print(echo('1'))
```

In Python, the type of a variable is **implicitly declared.** Even there is no type specification, functions can still accept references and returns a reference.

There are two main techniques you can use to ensure a function operations well regardless of input types.

The first approach is chosen by C++, you need to cover all possibilities, such as multiple add() functions for float, int, string addition. It is troublesome as programmers need to know and implement all the possible solutions.

The second approach is selected by Python. You just **ask the data itself (built-in) to perform the operation**. For example, when you execute c = a + b, Python executes c = a.__add__(b). However, not all functions are defined by built-in types, such as __len__() of integer is not defined. Python uses "**duck typing**", as the saying goes "If it looks like a duck, walks like a duck, and quacks like a duck, it's a duck". For example,

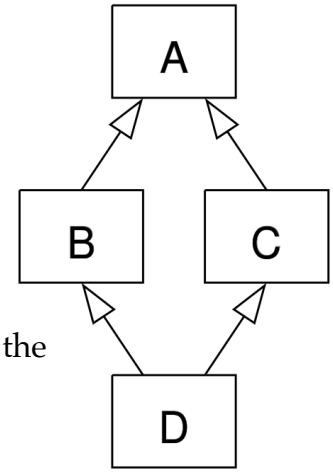
```
class Phone:
        def __init__(self, phone):
                self.phone = phone
        def boot(self):
                self.phone.boot()
class iPhone:
        def boot(self):
                print("iPhone booting")
class S9:
        def boot(self):
                print("S9 booting")

p1 = Phone(iPhone())
p2 = Phone(S9())
p1.boot()
```

Output:

```
iPhone booting
S9 booting
```

Although “iPhone” and “S9” are of different types, they both provide same operation, so they can be used to build a Phone object. Similarly, you can use an object to invoke common methods and variables of different types of objects.

v. Deadly Diamond of Death Problem



Wikipedia: The "diamond problem" problem is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?

Python's solution: The order of inheritance affects the class semantics. Children precede their parents and if a class inherits from multiple classes, they are kept in the order specified in the tuple of base classes. Thus, the method resolution order is: D, B, C, A.

To make it clear, let's look at the following example:

```
class A:
        def show(self):
                print("A")
class B(A):
        def show(self):
                print("B")
class C(A):
        def show(self):
                print("C")
class D(B, C):
        pass

D().show()
```

Output:

```
B
```

If we change "D(B, C)" to "D(C, B)", we will get output "C".
If we don't implement show() in class B, we will get output "A" with Python2 and "C" with Python3.

Exercises

Let's create an ordering system for a shop that sells smoothie. It allows customers to customise their own smoothie by mixing a number of ingredients.

There are 3 possible ingredients you can add to create a smoothie. When an ingredient is added, it contributes a certain amount of volume in the smoothie.

Ingredient	Volume
Apple	50 - 100ml
Orange	150 - 200ml
Milk	100 - 150ml

There are six python files, including Shop.py, Smoothie.py, Ingredient.py, Milk.py, Apple.py, Orange.py. They all are objects and are separated as modules.

Section 1. Ingredient.py

Apple, Orange, and Milk are ingredients, thus we can create a superclass to contain the common information and behaviours of them. There are two classes in Ingredident.py, namely **Ingredient** and **IngredientCreator**. **Ingredient** contains information of an ingredient and **IngredientCreator** is used to generate Ingredient objects.

- We need to generate random numbers, since the volume of ingredients may be different.

```
import random
```

- We need to import **abc** module (Abstract Base Class). Abstract methods may not be implemented in the base class, subclasses will be forced to override the implementation.

```
from abc import *
```

- We can declare both classes as abstract classes by inheriting from ABC.

```
class Ingredient(ABC):
class IngredientCreator(ABC):
```

- In class **Ingredient**, getName() is an abstract method. It is overridden in subclasses. The **pass** statement is used when a statement is required but you don't want anything to execute.

```
@abstractmethod
def getName(self):
        pass
```

- In class **IngredientCreator**, createIngredient() and getName() are abstract methods, since different ingredient creators have different implementations.

```
@abstractmethod
def createIngredient(self):
   pass
@abstractmethod
def getName(self):
   pass
```

Section 2. Milk.py, Orange.py, Apple.py

- First of all, they are inherited from classes in Ingredient.py, so we need to import Ingredient module.

```
from Ingredient import *
```

- In class **Milk**, when we create a Milk instance, we need to invoke Ingredient.__init__() to set the volume.

```
def __init__(self):
    super().__init__(100, 150) #min, max volume
```

- In class **Milk**, we need to override getName() to return "Milk".

```
def getName(self):
    return "Milk"
```

- In class **MilkCreator**, we need to override getName() to return "Milk".

```
def getName(self):
    return "Milk"
```

- In class **MilkCreator**, we need to override createIngredient(), then it creates a new Milk instance when it is called.

```
def createIngredient(self):
    return Milk()
```

- Now try to complete **Orange.py** and **Apple.py**!

Section 3. Smoothie.py

- First of all, import modules that you may need.

```
from Milk import *
from Orange import *
from Apple import *
from Ingredient import *
```

There are two methods: **addIngredient(self, newIngred)** which accepts an Ingredient object and adds to the list **ingredients**, and **printInfo(self)** which prints out the Name and Volume of ingredients added.

```
def addIngredient(self, newIngred):
    self.ingredients.append(newIngred)
    print("Added " + {name} + " " + {volume} + "ml")
    self.totalAmount = self.totalAmount + {volume}
def printInfo(self):
    print("Smoothie ingredients: “)
    for i in range(len(self.ingredients)):
        print({name}+ " (" + {volume} + "ml)")
```

☺You need to replace {name} and {volume} by correct implementations.

Hints:

1. You need to use **self.** to access class variables.
2. **getName()** and **getVolume()** can be used to get Ingredient name and volume!

Section 4. Shop.py

```
class Shop:
    ingredients = ???
    smoothie = ???

    def __init__(self):
        pass

    def askForIngredients(self):
        while True:
            for i in range(len(self.ingredients)):
                print(str(i + 1) + '.' + self.ingredients[i].getName())

            print("What would you like to add to your smoothie?")

            try:
                choice = int(input("Please enter your choice (1-" + str(len(self.ingredients)) + ",
or 0 to finish the order): "))

                if choice == 0:
                    break
                elif choice > 0 and choice < len(self.ingredients) + 1:
                    ingredient = self.ingredients[choice - 1].createIngredient()
                    self.smoothie.???
            except:
                print("Error occured.")

    def start(self):
        self.askForIngredients()
        self.smoothie.printInfo()

Shop().start()
```

1

2

3

1. **ingredients** is a list that contains ingredient creators, so we can invoke **self.ingredients[xxx].createIngredient()** to generate ingredient and put in smoothie. There are three ingredient creators in total: AppleCreator, OrangeCreator, MilkCreator.

```
ingredients = [AppleCreator(), OrangeCreator(), MilkCreator()]
```

2. **smoothie** is a Smoothie object that contains information of ingredients.

```
smoothie = Smoothie()
```

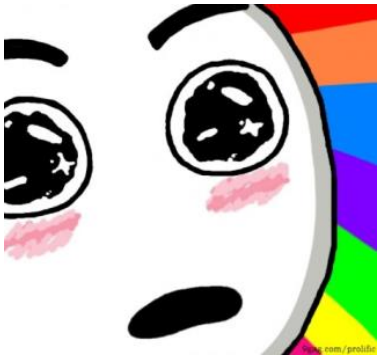
3. We need to add ingredient into smoothie.

```
self.smoothie.addIngredient(ingredient)
```

Remember **polymorphism**? Apple (AppleCreator), Orange (OrangeCreator), Milk (MilkCreator) are subclasses of Ingredient (IngredientCreator), and Ingredient (IngredientCreator) has abstract methods, we can use them to access subclasses regardless of their types.

- Now save the files, try to run the code and see if it works ☺?

```
python3 Shop.py
```



Congratulations! You have learned the basics of OOP in Python! Well done and thanks for your support ☺. You can check the completed code in directory **exercise-finished/**.

References

1. https://docs.python.org/3/reference/datamodel.html#basic-customization
2. http://www.cs.toronto.edu/~david/courses/csc148_f15/content/adts/stack_details.html
3. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/index.htm
4. https://sites.google.com/site/ubceece310/
5. http://www.expertphp.in/article/concept-of-abstraction-in-php
6. https://pythonspot.com/encapsulation/
7. https://www.programiz.com/python-programming/inheritance
8. https://www.python-course.eu/python3_multiple_inheritance.php
9. https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem
10. http://blog.thedigitalcatonline.com/blog/2014/08/21/python-3-oop-part-4-polymorphism/
11. Python Crash Course: A Hands-On, Project-Based Introduction to Programming
12. COMP2396 Lecture Notes