THE UNIVERSITY OF HONG KONG

FACULTY OF ENGINEERING

Department of Computer Science
COMP2123 Programming Technologies and Tools
Group-based self-learning report

Object-Oriented Programming and Python

Group Members	UID
1. Cai Yuxi	3035446887
2. Dong Xinhang	3035449217
3. Tao Yufeng	3035447049

COMP2123 Programming Technologies and Tools

Lab 8.1 Basic Object-Oriented programming

Estimated time to complete this lab: 30 mins

Objectives

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

- Understand what is Object-Oriented programming
- Use `class` to define objects in Python

Preparations

1. Download the sample files `lab8.tar` from Moodle.
   a. Browse to the course Moodle page (http://moodle.hku.hk).
   b. **Right click** the link of `lab8` Chapter 8 Lab materials > Save Link As
   c. Save the materials to your home directory
2. Extract the sample files to ~\\`lab8`.
   a. Open file explorer Applications > Accessories > Files
   b. Browse to your home directory.
   c. Issue the command "`tar -xvf lab8.tar`".
3. Open Shell
   a. Application > Terminal Emulator
   b. Browse to directory `~/lab8`

```
$ cd ~/lab8/
```

   c. There are 3 directories created in the `lab8`.

```
$ ls
3 directories
```

Section 1. What is object-oriented programming

1. Concepts

- In real world, "object" means tangible things that we can feel, or operate. However, in programming, objects are models of somethings that have certain behaviors. Formally, an object is a collection of **data** and associated **behaviors**.
- Object-oriented Programming, or *OOP* for short, is a programming paradigm which provides a means of structuring programs so that properties and behaviors are bundled into individual *objects*.

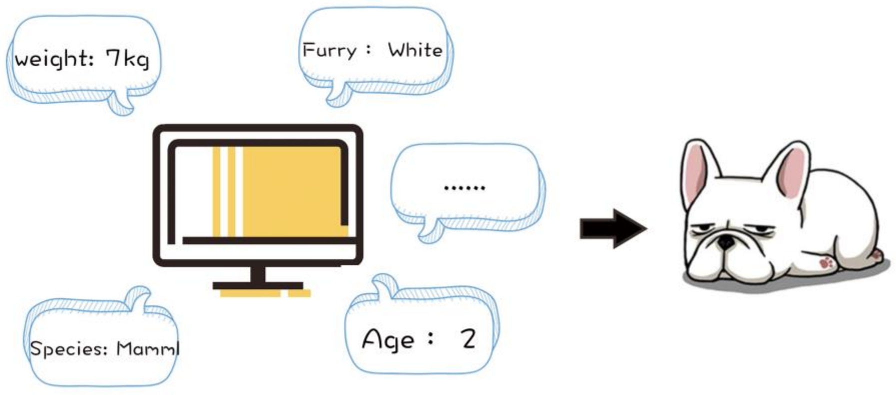


Figure: Structure a program to bundle the properties into individual objects

2. How do we distinguish two different types of objects in programming?

- How can you represent a dog in programming?
  -You can build a `structure`, which contains different elements such as its age, name and weight, etc.

```
Name=Snowy
Age=2
Weight=7
......
```

Figure: Build a structure to define a dog

How about representing 100 dogs?

Use `classes` and `instances`!

- `Classes` are used to create new user-defined data structures that contain arbitrary information about objects.
- The procedure of using classes to define objects is:
  - **i.** Define a new class
  - **ii.** Initiating Objects

Section 2. How to build a class

1. How to define a class

Suppose we are going to write a program to find the heaviest dog.

Let's first create a program called `findTheHeaviest.py`

```
$ cd ~/lab8/classes
$ gedit findTheHeaviest.py
```

1) Define a simplest class

The simplest class in python is like:

```
#!/usr/bin/python
class dog:
pass
```

The keyword `pass` is a placeholder allowing us to run the code without throwing error. In this way, we can define an empty class and run the program successfully.

2) Attributes

- All objects have characteristics, which are called `attributes` in class. There are generally two kinds of attributes: `class attributes` and `instance attributes`.

a) Class attributes

Class attributes are the same for all instances, for example, all of the dogs in our program are animals rather than plants.

```
class dog:
    #class attributes
    specials = “animal”
```

b) Instance attributes

- `Instance attributes` are characteristics of objects that different from each other. Such as the age, weight and name of dogs in our program

```
class dog:
   #class attributes
   specials = “animal”
   #Instance attributes
   def __init__(self, name, age, weight):
        self.name = name
        self.age = age
        self.weight=weight
```

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

The class can be given a special method called __init__() which is run when the class is instantiated, receiving the arguments passed when calling the class; the general name of such a method, in the OOP context, is *constructor*, even if the __init__() method is not the only part of this mechanism in Python.

Note: The first argument in __init__() should always be “self”!

2. Instantiating Objects

- If we say that the class is the skeleton, or the blueprint of the objects, an instance is the class with the actual value. In our example, it is not a dog who has name, age and different weight anymore. It can become a real 2-year-old dog Snowy, which weights 7 kilograms.
- **Instantiating means that we make a new instance of the class**

```
class dog:
#The same as what is defined above
Snowy =dog(“Snowy”2,7)
Snoopy =dog(“Snoopy”,8,6)
```

- **Note: The arguments in the parentheses are corresponding to what you defined in the class.**

3. Using objects in functions

- After defining the class and making some new instances, let’s set up a function to compare the weight of the two dogs in our instance.

```
    class dog:
       #The same as what is defined above
    Snowy =dog(“Snowy”, 2, 7)
    Snoopy =dog(“Snoopy”, 8, 6)
    def heaviest(*args):
       return max(args)
print("The heaviest one weights
{} Kilograms.".format(heaviest(Snowy.weight,Snoopy.weight)))
```

- Let’s run the script and see what will happen! ☺

```
$ ./findTheHeaviest.py
   The heaviest one weights 7 Kilograms.
```

Reference:

Python Documentation. Basic Costomization.

https://docs.python.org/2/reference/datamodel.html?highlight=__init__#basic-customization

Object-Oriented Programming in Python 3

https://realpython.com/python3-object-oriented-programming/

OOP concepts in Python 2.x - Part 1

http://blog.thedigitalcatonline.com/blog/2014/03/05/oop-concepts-in-python-2-dot-x-part-1/

COMP2123 Programming Technologies and Tools

Lab 8.2 Four major principles of Object-Oriented Programming in Python

Estimated time to complete this lab: 1 hour

Objectives

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

- Know what are the four key principles of Object-Oriented Programming
- Understand how the four principles are embodied in Python

Having learned the above basic concepts, it's time to introduce the famous "four pillars of Object-Oriented Programming (OOP)": Encapsulation, Data Abstraction, Inheritance and Polymorphism. They are four key principles that OOP follows, and this section will briefly cover how they are embodied in Python.

Section 1. Encapsulation

Generally speaking, encapsulation is the mechanism for restricting the access to some of an object's components, i.e. data hiding. This means that the internal representation of an object can't be seen from outside of the object's definition.

However, recalling that in Python, there are no keywords like "public" and "private" to define the accessibility. In other words, Python requires that all attributes are public.

Question: How can I achieve data hiding in Python?

- Don't worry! There is a method in Python to define private attributes:
  Using underscore as prefix (i.e. single "_" or double "__" in front of the variables and function names) can hide them when they are accessed outside of the class.

- Let’s browse to `~/lab8/principles` and open the file called `encapsulation.py`

```
$ cd ~/lab8/principles
$ gedit encapsulation.py &
```

```
1.  #!/usr/bin/python
2.  class Computer:
3.      def __init__(self):
4.          self.__maxprice = 900
5.      def sell(self):
6.          print("Selling Price:{}".format(self.__maxprice))
7.  c = Computer()
8.  c.sell()
9.  # change the price
10. c.__maxprice = 1000
11. c.sell()
```

Code Explanation

- **Line 2-6**: we created a class named `Computer` together with its instance attribute `__maxprice` (to store the maximum selling price of the computer) and method `sell` (to display the maximum price). Note that a double “__” was used to achieve data hiding.
- **Line 7-8**: we instantiated the class to be `c` and called the instance method to print the original selling price.
- **Line 10-11:** We tried to modify the price by accessing the private attribute directly.
- Run `encapsulation.py`

```
$ ./encapsulation.py
Selling Price: 900
Selling Price: 900Selling Price: 900
```

Not surprisingly, we failed to modify because of the encapsulation principle: Python treats the `__maxprice` as private attributes ☹

To change the value, we can add a setter function i.e. `setMaxPrice()` which takes price as parameter.

Please add the following right after line 6 (Be careful of the indentation!):

```
def setMaxPrice(self, price):
        self.__maxprice = price
```

Please add the following at the end (Be careful of the indentation!):

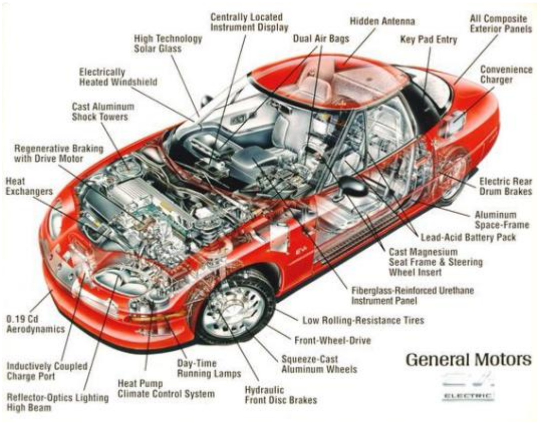
```
c.setMaxPrice(1000)
c.sell()
```

- Run again and you should see that modification is successful ☺

```
$ ./encapsulation.py
Selling Price:  900
Selling Price:  900
```

Section 2. Abstraction

Think of a car!


Centrally Located Instrument Display
Hidden Antenna
All Composite Exterior Panels
High Technology Solar Glass
Dual Air Bags
Key Pad Entry
Electrically Heated Windshield
Convenience Charger
Cast Aluminum Shock Towers
Regenerative Braking with Drive Motor
Heat Exchangers
Electric Rear Drum Brakes
Aluminum Space-Frame
Lead-Acid Battery Pack
Cast Magnesium Seat Frame & Steering Wheel Insert
Fiberglass-Reinforced Urethane Instrument Panel
Low Rolling-Resistance Tires
0.19 Cd Aerodynamics
Front-Wheel-Drive
General Motors
Inductively Coupled Charge Port
Day-Time Running Lamps
Squeeze-Cast Aluminum Wheels
Heat Pump Climate Control System
Hydraulic Front Disc Brakes
Reflector-Optics Lighting High Beam
ELECTRIC

Most likely you will get the left one rather than the right one…

- Data abstraction is the simplest principle to understand. Through the process of abstraction, a programmer hides all but the relevant data about an object in order to reduce complexity and increase efficiency.

- Abstraction and encapsulation are tightly related in a sense that abstraction is achieved through encapsulation. Just like the car, we only need to know that the gas is on the right, the break is on the left, put it in drive and the steering wheel tells the car where to go. This is abstraction. We are only shown the simplest possible interface and the non-essential information to that interface has been hidden.

Section 3. Inheritance

Inheritance is a way of creating a new class for using details of an existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class)

Let's open the file called `inheritance.py`

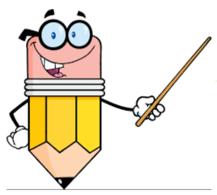
```
$ gedit inheritance.py &
```

```
#!/usr/bin/python
# parent class
class Bird:
    def __init__(self):
        print("Bird is ready")
    def whoisThis(self):
        print("Bird")
    def swim(self):
        print("Swim faster")
# child class
class Penguin(Bird):
    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")
    def whoisThis(self):
        print("Penguin")
    def run(self):
        print("Run faster")
peggy = Penguin()
```

```
21. peggy.whoisThis()
22. peggy.swim()
23. peggy.run()
```

Code Explanation

- **Line 3-19**: we created a parent class named `Bird` and a child class named `Penguin`.



Use super()function before the __init__()method so that we can pull the content of __init__()method from the parent class into the child class.

- **Line 20**: we instantiated the child class to be `peggy`. Let's run the script to see what will happen:

```
$ ./inheritance.py
Bird is ready
Penguin is ready
Penguin
Swim faster
Run faster
```

- **Line 20:** at the time of instantiation, the `super()` built-in function returns a proxy object that allows you to refer the parent class by "`super`". Therefore, we first called `__init__`method of the `Bird` class (from the `Penguin` class) generating the first output line, followed by its own printing method generating the second output line.
- **Line 21:** the child class modified the behavior of the parent class demonstrated by the behavior of `whoisThis` method. Although the name is overlapped, the implementation follows the child class definition.
- **Line 22:** the child class inherited the functions of its parent class. The `Penguin` class can use the `swim()` method defined in the `Bird` class.
- **Line 23:** furthermore, we extend the functions of parent class, by creating a new `run()` method. One of the most powerful features of inheritance is the ability to extend components without any knowledge of the way in which a class was implemented.

Section 4. Polymorphism

Polymorphism is an ability (in OOP) to use common interface for multiple form (data types). Functions or methods that are named the same thing can do things differently. This makes our code easier to interact with ☺

Let's open the file called `polymorphism.py`

```
$ gedit polymorphism.py &
```

```
1.  #!/usr/bin/python
2.  class Parrot:
3.      def fly(self):
4.          print("Parrot can fly")
5.      def swim(self):
6.          print("Parrot can't swim")
7.  class Penguin:
8.      def fly(self):
9.          print("Penguin can't fly")
10.     def swim(self):
11.         print("Penguin can swim")
12. # common interface
13. def flying_test(bird):
14.     bird.fly()
15. #instantiate objects
16. blu = Parrot()
17. peggy = Penguin()
18. # passing the object
19. flying_test(blu)
20. flying_test(peggy)
```

Code Explanation

- **Line 2-11:** we created two classes named `Parrot` and `Penguin` respectively. Each of them have common method `fly()` method. However, their functions are different.

- **Line 13-14:** to allow polymorphism, we created a common interface i.e. `flying_test()` function that can take any object. You can try different object names in the definition other than `bird` and they also work, because it is just a placeholder.
- **Line 19-20:** we passed the objects `blu` and `peggy` in the `flying_test()` function. Let's test whether it ran effectively:

```
$ ./polymorphism.py
Parrot can fly
Penguin can't fly
```

Hence, we verified that polymorphism allows us to use two different class types, in the same way, achieving each of their outcomes.

Reference:

Python Object Oriented Programming:

https://www.programiz.com/python-programming/object-oriented-programming#key-points

Python super():

https://www.programiz.com/python-programming/methods/built-in/super

The Four Principle of Object Oriented Programming:

https://medium.com/@benjaminpjacobs/the-four-principle-of-object-oriented-programming-f78600f62608

COMP2123 Programming Technologies and Tools

Lab 8.3 Iterator Protocol

Estimated time to complete this lab: 1 hour

Objectives

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

- Understand fundamental iterator protocol in python.

Section 1. Introduction: for-loop

Through the previous study in COMP2123, we have learned that we can loop over most container objects using a **for** statement.

Consider for1.py as an example:

```
$ cd ~/lab8/iterator
$ gedit for1.py &
```

```
#!/usr/bin/python
numbers = [1, 2, 3]
for n in numbers:
    print(n)
```

Run for1.py

```
$ chmod u+x for1.py
$ ./for1.py
1
2
3
```

How does the loop fetch individual elements from the object it is looping over? And how can we support the same programming style in the user-defined objects?

We will find the answers to these questions after we learn the Python's iterator protocol in this lab! ☺

Section 2: Create an iterator that iterates forever

1. What is iterator?

An **iterator** is an object that can be iterated upon, which means we can traverse through all the values. In python, an iterator is an object which adopts the iterator protocol. It includes the methods **__iter__( )** and **next( )**.

2. Iterable objects

An **iterable object** is an object that implements __iter__, which is expected to return an iterator object. Lists, tuples, dictionaries and sets are all **iterable objects**.

- Let's consider hello.py as an example. (Please try to execute the following code first. The details will be discussed later. ☺)

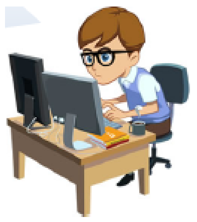
```
$ gedit hello.py &
```

```
#!/usr/bin/python
class Repeater:
def __init__(self, word):
    self.word = word
def __iter__(self):
    return self
def next (self):
    return self.word
repeater = Repeater('Hello')
for item in repeater:
    print(item)
```

```
$ chmod u+x hello.py
$ ./hello.py
```

What is your output?

Oops… A lot of 'Hello' printed on my screen and the loop seems never ending…



(We can use *Ctrl* + *C* to break out of the infinite loop.)

- Let's take a closer look at the above program.

```
class Repeater:
def __init__(self, word):
    self.word = word
```

  - As introduced in the previous section, when we create a class, the method called __init__ allows us to initialize the object, which is similar to the constructor we use when defining a class in C++.

```
def __iter__(self):
    return self
```

  - The __iter__method is what makes an objects iterable. It returns the iterator object itself. In other words, calling __iter__ returns an object that implements the next method. So actually, the method does not necessarily return itself. It can return any object with next method.

```
def next (self):
    return self.word
```

  - The next method allows us to do operations and returns the next item in the sequence. In our example, the method will simply return the word 'Hello'.

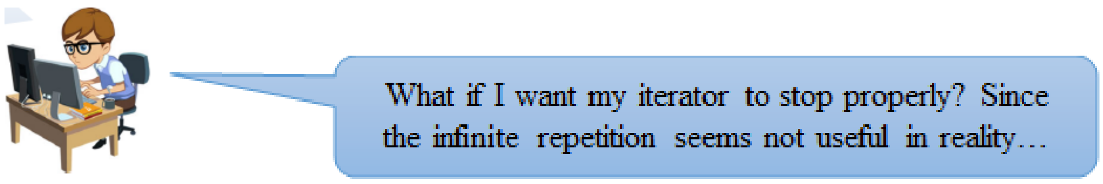
- To facilitate your understanding in the working mechanism of for-loop, the following codes just demonstrate the rationale behind.

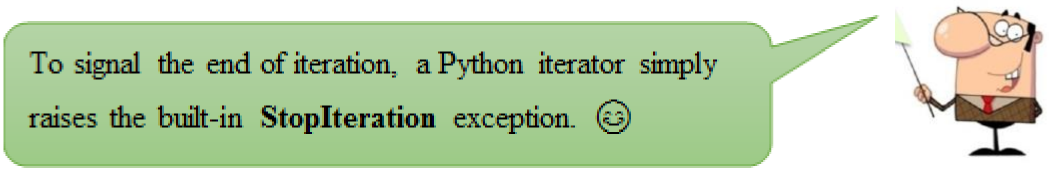
```
repeater = Repeater('Hello')
iterator = repeater.__iter__()
while True:
    item = iterator.next()
    print(item)
```

As you can see, the never- ending for-loop actually does the following step:

- It first prepared the repeater object for iteration by calling its __iter__ method, which returned the actual iterator object.
- Then, the loop repeatedly calls the iterator object's next() method to retrieve values from it.

Section 3: Create an iterator that can stop properly


What if I want my iterator to stop properly? Since the infinite repetition seems not useful in reality…


To signal the end of iteration, a Python iterator simply raises the built-in **StopIteration** exception. ☺

```
raise StopIteration
```

- We can use the above statement to signal that there are no more values available to iterate over and the for-loop will terminate automatically.
- Let's change the previous program to make it stop printing 'Hello' after a certain number of repetitions.

```
$ gedit boundedRepeater.py &
```

```
#!/usr/bin/python
class BoundedRepeater:
def __init__(self, word, max_times):
    self.word = word
    self.max_times = max_times
    self.count = 0
```

- In __init__ method, the variable max_times helps to specify the number of iterations we want to carry out and the variable count will record the number of iterations it has done.

```
def __iter__(self):
    return self
def next(self):
    if self.count >= self.max_times:
        raise StopIteration
    self.count += 1
    return self.word
```

- Now each time the next method is called, the for-loop will check if-statement and raise StopIteration to terminate the iteration when the condition is not satisfied.

```
repeater = BoundedRepeater('Hello', 3)
for item in repeater:
    print(item)
```

- Let's try to execute this program.

```
$ chmod u+x boundedRepeater.py
$ ./boundedRepeater.py
Hello
Hello
Hello
```

Now we can get exactly three repetitions of 'Hello'. ☺

Section 4: Create an iterator that can reverse the input

After understanding the iterator protocol above, we can easily create an iterator that can implement behaviors we desire.

- Let’s consider the Reverse.py example, which can reversely output the word we input.

```
$ gedit Reverse.py &
```

```
#!/usr/bin/python
class Reverse:
def __init__(self, word):
   self.word = word
   self.index = len(word)
```

- ○ In python, len(s) can return the number of items in an object s where s can be a sequence (string, byte, tuple, list or range) or a collection (dictionary or set).

```
def __iter__(self):
return self
def next (self):
   if self.index == 0:
       raise StopIteration
   self.index = self.index - 1
   return self.word[self.index]
```

- ○ The next method will return the character of the input string reversely and terminate the iteration once all the characters have been printed.

```
reverse = Reverse(‘Hello’)
for item in reverse:
   print (item),
```

- Let’s try to execute the program and check whether the word is reversely printed.

```
$ chmod u+x Reverse.py
$ ./Reverse.py
o l l e H
```

It works! ☺

Congratulations! You have finished this self-learning lab about Object-oriented programming and Python. Keep exploring more!

References:

Python classes

https://docs.python.org/2/tutorial/classes.html

Python iterator tutorial

https://dbader.org/blog/python-iterators