CSE4IP
Lecture 11 –Classes and Objects

Ayad Turky

Last time

- We use files to store data and programs
- Data on files are known as persistent data because they exist between execution of programs
- There are two kinds of files: text file and binary file
- Writing to a file consists of three main steps: Open the file, Write to the file and Close the file
- To open a file for reading involve 3 general steps: Open the file for reading, Read the file (and process data) and Close the file (when the reading is finished)

The Concept of Object and Class

- The concept of object in programming is very similar to that in everyday language
- Expectedly, it is a very general concept
- Essentially, an object is something that is distinguishable from other things
- We use this term to refer to, for example, people and organizations, to physical things like tables and chairs, and to non-tangible things like accounts, contracts, etc.

The Concept of Object and Class

- A class is template or a blueprint for making objects
- We understand the relationship between class and object intuitively and accurately, as shown in the following example
- Suppose you are in a restaurant, and you call a waiter over, and pointing to the dish of the customer on the next table, you say:

  "I want that dish"
- What do you expect the waiter to do?

The Concept of Object and Class

Should he

a) Go to the next table and take the dish and bring it back to you? or

b) Go to the kitchen and ask the cook to prepare a similar dish?

- If (a) is the intended action, then "that dish" has been used in the sense of an object
- If (b) is the intended action, then "that dish" has been used in the sense of a class
- In this sense, "that dish" represents a type of dish, serving as a "blue print" for individual dishes of that type

The Concept of Object and Class

- In programming, we can be more specific about what we mean by an object
- An object is an entity that combines both data and behavior
  - The data held by an object is known as its data attributes (or attributes for short)
  - The behavior is represented by its methods that can access the object's data attributes

The Concept of Object and Class

- Q1: How can such a concept be implemented?
- Q2: How can such a concept be useful for problem solving?

The Process of Defining Classes, and Creating and Using Objects

Class

- In real life, the concept of object may precede the concept of class
- In object-oriented programming, the same thing may happen in the mind of a programmer.
- But in the program itself, the class must be defined first,

  Then the class's definition is used as a template to create the objects and then to interact with them

How to define a class

- Consider, for example, the bank accounts
- Each account is an object
- Each would have some data such as the account number, the customer's name, the balance, etc.
- In addition, an account object also have a number of associated operations

  These operations allow us to deposit money into the account, or to withdraw some money, or to make inquiry about the balance.

How to define a class

- Those characteristics (data and operations) are to be captured in the class definition
  - The data are also known as data attributes or simply attributes
  - The operations are implemented as methods

- To define a class is to define its attributes and methods

How to create objects

- Each class will have one special method which is referred to as the "constructor"
- We use the constructor methods to create objects. In an application, some objects must be initially created in the main method of a class

- We create objects with the constructor

How to interact with objects

- We interact with an objects
  - by accessing its attributes or
  - by applying its methods
- We also say that we send messages to it
- A message requests the object
  - To provide us with some information it has, or
  - To execute a method
- We interact with objects by sending messages to them

How to interact with objects

- We interact with an objects
  - by accessing its attributes or
  - by applying its methods
- We also say that we send messages to it
- A message requests the object
  - To provide us with some information it has, or
  - To execute a method
- We interact with objects by sending messages to them

Example to Illustrate How to Define Classes, Create Objects and Use Object: Bank Account

Example: Bank

- Assume that for a bank account, attributes of interest are
  - the account number
  - the customer's name
  - the balance
- And the operations of interest are
  - to make a deposit
  - to make a withdrawal
  - to get the balance

Example: Bank

- We will illustrate the process in several steps
- First, we define a version 1
- Then we add more features to it
- And we will see what we can achieve at each stage

Version 1

```
class BankAccount:

    def __init__(self, accountNr, customerName, balance = 0):
        self.accountNr = accountNr
        self.customerName = customerName
        self.balance = balance
```

Version 1

- We start the definition with the keyword class followed by the name of the class
- Then we have a special method with the name

__init__

  - This name means that it is the constructor
  - It is the method that will be used to create BankAccount objects
  - This method is also known as the initializer

Version 1

- The constructor has a special parameter

  self

  – This special parameter stands for represents the current object

- In addition, the constructor has another three parameters: accountNr, customerName and balance

Version 1

- The constructor has these three statements

```
self.accountNr = accountNr
self.customerName = customerName
self.balance
```

which means

- The object has attributes named **account**Nr, **customerName** and **balance**,
- and they are **initialized** to the values of parameters accountNr, customerName and balance, respectively

Create and Use BankAccount objects

```
>>> a1 = BankAccount("A10", "bob", 100)
>>> a1.accountNr
'A10'
>>> a1.customerName
'Bob'
>>> a1.balance
100
```

Create and Use BankAccount objects

```
>>> a2 = BankAccount("A20", "Alice")
>>> a2.accountNr
'A20'

>>> a2.customerName
'Alice'

>>> a2.balance
0
```

Version 2

- In the previous version, when we display object itself, we don’t see any details about the attributes

- In this version, we fix it with the special method

__repr__

Version 2

```
class BankAccount:
    def __init__(self, accountNr, customerName, balance = 0):
        self.accountNr = accountNr
        self.customerName = customerName
        self.balance = balance

    def __repr__(self):
        return "BankAccount<nr: " + str(self.accountNr) +  \
           ", name: " + str(self.customerName) +    \
           ", balance: " + str(self.balance) + ">"
```

Version 2

```
>>> a1 = BankAccount("A10", "Bob", 1000)
>>> a1
BankAccount<nr: A10, name: Bob, balance: 1000>


>>> print("Bob's account:", a1)
Bob's account: BankAccount<nr: A10, name: Bob, balance: 1000>
```

Version 3

- Define methods
  - To deposit
  - To withdraw

Version 3

```
class BankAccount:
    def __init__(self, accountNr, customerName, balance = 0):
        self.accountNr = accountNr
        self.customerName = customerName
        self.balance = balance

    def __repr__(self):
        return "BankAccount<nr: " + str(self.accountNr) +  \
            ", name: " + str(self.customerName) +          \
            ", balance: " + str(self.balance) + ">"

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
         self.balance -= amount
```

Version 3

```
>>> a = BankAccount("A10", "Bob", 1000)

>>> a
BankAccount<nr: A10, name: Bob, balance: 1000>

>>> a.deposit(200)
>>> a
BankAccount<nr: A10, name: Bob, balance: 1200>

>>> a.withdraw(100)
>>> a
BankAccount<nr: A10, name: Bob, balance: 1100>
```

Example: The Digital Clock

- We model a digital clock
- It keeps the time with attributes
  - hours
  - minutes
  - seconds
- It has method tick, which increase the time by on second

Example: The Digital Clock

```
class DigitalClock():

    def __init__(self, hours =0, minutes = 0, seconds = 0):
            self.hours = hours
            self.minutes = minutes
            self.seconds = seconds

     def __repr__(self):
            return "Clock<hours:" + str(self.hours)    \
                    + ",minutes:" + str(self.minutes)  \
                    + ",seconds:"+ str(self.seconds) + ">"
```

Example: The Digital Clock

```
def tick(self):
        self.seconds += 1

        if self.seconds == 60:
            self.seconds = 0
            self.minutes += 1

        if self.minutes == 60:
            self.minutes = 0
            self.hours += 1

        if self.hours == 24:
            self.hours = 0
```

Example: The Digital Clock

- Test 1 – How tick works

```
c = DigitalClock()
print(c)
for _ in range(5):
    c.tick()
    print(c)
```

Example: The Digital Clock

- Test 2 – To see how minutes and hours change

```
c = DigitalClock(23, 59, 58)
print(c)
for _ in range(5):
    c.tick()
    print(c)
```

Example: The Digital Clock

- Test 3 – Increate seconds when 1 second passes, using function sleep

```
import time
c = DigitalClock(23, 59, 55)

for _ in range(10):
        time.sleep(1)      # sleep for 1 second
        c.tick()
        print(c)
```

Example: The Digital Clock

- Test 4 – Display time at same spot, simulating a physical clock

```
import time

c = DigitalClock(23, 59, 55)

for _ in range(10):
    time.sleep(1)   # sleep for 1 second
    c.tick()
    print("\r{:02d}:{:02d}:{:02d}".
                format(c.hours, c.minutes, c.seconds),
                end ="")
```

Advantages of object-oriented approach

- This example demonstrates the advantages of object-oriented approach:
  - First, we concentrate on modeling the digital clock. In this example, we simply focus on the attributes and the **tick** method
  - Once the class is available, we can use it in many different ways
- To reiterate,
  - We separate **modelling** from **use**
  - We can **reuse** the same class in many different ways

Example: The Millionaire Project

- You have a plan to be a millionaire,
  - In the first month, you save 1 dollar
  - In the second month,2 dollars
  - In the third month, 4 dollars, etc.
  - Each month, you save twice as much the amount for the previous month.
- You may be interested in a number of questions,

  e.g.
  - How much will you save after 1 year?
  - When will you become a millionaire?

Example: The Millionaire Project

- You can look at this as a project, the saving project
- The state of your project changes from month to month
- The following information can characterize its state:
  - What month you are in?
  - How much you save this month?
  - How much is the total amount that you have save up to this month?
- Armed with this conception, you can model the project

Example: The Millionaire Project

```
class SavingProject:
    def __init__(self):
            self.month = 1
            self.saved = 1
            self.total = 1

    def next(self):
            self.month = self.month + 1
            self.saved = self.saved * 2
            self.total = self.total + self.saved

    def __repr__(self):
            return "<SavingProject: month:" + str(self.month)   \
                + "/saved:" + str(self.saved)    \
                + "/total:" + str(self.total) + ">"
```

Test 1

- Create a SavingProject and apply next 5 times

```
sp = SavingProject()
print(sp)
for _ in range(5):
    sp.next()
    print(sp)
```

Test 2 – How much saved in 12 months?

```
sp = SavingProject()
print(sp)
for _ in range(11): # repeat 11 times
    sp.next()
    print(sp)
print("Total amount saved after 12 months:", sp.total)
```

Test 3: When will be a millionaire?

```
sp = SavingProject()
while sp.total < 1E6:
    sp.next()
    print(sp)  # for inspection
    print("Number of months it takes to be a millionaire:", sp.month)
```

Example

- We separate modeling and use
- Defining class SavingProject is straight forward (the main work is for method **next**)
- Once the class us available,
  - We can test it and see exactly what happens from month to month (test 1).
  - We can use the class to answer questions we may interested in (tests 2 and 3).

Storing class in Modules (How source code is organized for practical projects)

Storing class in Modules

- For small examples, or when we are testing our ideas, we may put classes and use them in the same file
- In practice,
  - we often organize our classes by storing them in modules
  - and import the modules when we need them

Storing class in Modules

- As demonstration,
  - We put the definition of class DigitalClock in module digital_clock (a file with name digital_clock.py)
  - We import it into the program that simulates the display of time for a physical clock

Recap

- The concept of object in programming is very similar to that in everyday language
- Expectedly, it is a very general concept
- Essentially, an object is something that is distinguishable from other things
- A class is template or a blueprint for making objects

Thank you.

Be well.

latrobe.edu.au

La Trobe University
CRICOS Provider Code Number 00115M