Python 4: Object-Oriented Programming

금융공학 프로그래밍 I

About OOP

OOP is one of the major paradigms in programming, and nicely supported in Python

OOP has become an important concept in modern software engineering because

- It can help facilitate clean, efficient code (when used well)
- The OOP design pattern fits well with the human brain

OOP is supported in many programming languages:

- Python supports **both** procedural and object-oriented programming
- JAVA and Ruby are relatively pure OOP
- Fortran and MATLAB are mainly procedural, but with some OOP recently tacked on
- C is a procedural language, while C++ is C with OOP added on top

Procedural vs. Object-Oriented

- Procedural
  - The program has a state that contains the values of its variables
  - Functions are called to act on these data according to the task
  - Data are passed back and forth via function calls

- OOP
  - Data and functions are bundled together into "objects"

```
In [1]: x = [1, 5, 4]

In [2]: x.sort()

In [3]: x
Out[3]: [1, 4, 5]
```

Terminology

- Class
  - A blueprint for a particular class
  - It describes
    - What kind of data the class stores
    - What methods it has for acting on this data
- Object or Instance
  - A realization of the class, created from the blueprint
    - Each instance has its own unique data
    - Methods set out in the class definition act on this (and other) data
- Attributes
  - Attributes are accessed via "dotted attribute notation"

Defining Your Own Classes

```
In [1]: class Consumer:
   ...:     pass
   ...:

In [2]: c1 = Consumer()  # Create an instance

In [3]: c1.wealth = 10

In [4]: c1.wealth
Out[4]: 10
```

Consumer Class

```
class Consumer:

    def __init__(self, w):
        "Initialize consumer with w dollars of wealth"
        self.wealth = w

    def earn(self, y):
        "The consumer earns y dollars"
        self.wealth += y

    def spend(self, x):
        "The consumer spends x dollars if feasible"
        new_wealth = self.wealth - x
        if new_wealth < 0:
            print("Insufficent funds")
        else:
            self.wealth = new_wealth
```

Usage

```
In [1]: run consumer.py

In [2]: c1 = Consumer(10)

In [3]: c1.spend(5)

In [4]: c1.wealth
Out[4]: 5

In [5]: c1.earn(15)

In [6]: c1.spend(100)
Insufficent funds
```

```
In [2]: c1 = Consumer(10)

In [3]: c2 = Consumer(12)

In [4]: c2.spend(4)

In [5]: c2.wealth
Out[5]: 8

In [6]: c1.wealth
Out[6]: 10

In [7]: c1.__dict__
Out[7]: {'wealth': 10}

In [8]: c2.__dict__
Out[8]: {'wealth': 8}
```

self

- *self*
  - Any instance data should be prepended with *self*
  - Any method defined within the class should have *self* as its first argument
  - Any method referenced within the class should be called as *self.method_name*

- Details

```
In [6]: Consumer.__dict__   # Show __dict__ attribute of class object
Out[6]:
{'__doc__': None,
 '__init__': <function __main__.__init__>,
 '__module__': '__main__',
 'earn': <function __main__.earn>,
 'spend': <function __main__.spend>}
```

In fact the following are equivalent

- *c1.earn(10)*
- *Consumer.earn(c1, 10)*

Special Methods

```
class Foo:

    def __len__(self):
        return 42
```

Now we get

```
In [23]: f = Foo()

In [24]: len(f)
Out[24]: 42
```

```
class Foo:

    def __call__(self, x):
        return x + 42
```

After running we get

```
In [25]: f = Foo()

In [26]: f(8)   # Exactly equivalent to f.__call__(8)
Out[26]: 50
```

Account Class

```
class Account(object):
     num_accounts = 0
     def __init__(self,name,balance):
         self.name = name
         self.balance = balance
         Account.num_accounts += 1
     def __del__(self):
         Account.num_accounts -= 1
     def deposit(self,amt):
         self.balance = self.balance + amt
     def withdraw(self,amt):
         self.balance = self.balance - amt
     def inquiry(self):
         return self.balance

 # Create a few accounts
 a = Account("Guido", 1000.00)
 b = Account("Bill", 10.00)

 a.deposit(100.00)
 b.withdraw(50.00)
 name = a.name
```

Inheritance

- Inheritance Class

```
import random
class EvilAccount(Account):
    def inquiry(self):
        if random.randint(0,4) == 1:
           return self.balance * 1.10   # Note: Patent pending idea
        else:
           return self.balance

c = EvilAccount("George", 1000.00)
c.deposit(10.0)          # Calls Account.deposit(c,10.0)
available = c.inquiry()  # Calls EvilAccount.inquiry(c)
```

- __init__ for subclass

```
class EvilAccount(Account):
    def __init__(self,name,balance,evilfactor):
        Account.__init__(self,name,balance)    # Initialize Account
        self.evilfactor = evilfactor
    def inquiry(self):
        if random.randint(0,4) == 1:
           return self.balance * self.evilfactor
        else:
           return self.balance
```

Q & A

khhwang78@gmail.com