Programming in Python

13 – 객체지향 **Python** 프로그래밍

2016년 8월, 국민대학교 컴퓨터공학부

Python 과 OOP (Object-Oriented Programming)

- Python 의 큰 특징이 배우기 쉽다는 것이라던데, OOP 까지 꼭 신경써야 하는가?
  - 그럴 필요 없음, Python 은 쉽게 접근할 수 있는 스크립팅 언어라는 점이 매력적
  - 필요에 따라 작은 모듈들을 개발함으로써 쉽게 소프트웨어를 생산할 수 있음

- 그렇다면, 객체지향 Python 프로그래밍에 대해 알면 좋은가?
  - Python 은 모든 데이터를 객체로 관리하기 때문에, 객체지향 개념의 도입이 자연스러움
  - 공개 도메인에 존재하는 수많은 라이브러리들이 OOP 로 만들어져 있음
  - 프로그램의 규모가 커질수록 OOP 를 이용함으로써 얻을 수 있는 잇점도 따라 커짐
  - 배우기 어렵지 않음

생성자 (Constructor) 를 가지는 클래스

person.py

```
class Person:
        def __init__(self, name, job, pay):
                self.name = name
                self.job = job
                self.pay = pay
```

```
>>> from person import Person
>>> p = Person()
TypeError: __init__() takes exactly 4 arguments (1 given)

>>> p = Person('James', 'Developer', 100)
>>> p
<person.Person instance at 0x10189ab90>
>>> p.name
'James'
>>> p.job
'Developer'
>>> p.pay
100
>>> p.job = 'Manager'
>>> p.job
'Manager'
```

person.py

```
class Person:
        def __init__(self, name, job=None, pay=0):
                self.name = name
                self.job = job
                self.pay = pay
```

```
>>> from person import Person
>>> p = Person('James')

>>> p.name
'James'
>>> p.job
>>> p.pay
0
>>> p = Person()
TypeError: __init__() takes at least 2 arguments (1 given)
```

단위 테스트가 포함된 클래스 모듈

person.py

```
class Person:
        def __init__(self, name, job=None, pay=0):
                self.name = name
                self.job = job
                self.pay = pay

if __name__ == '__main__':
        bob = Person('Bob Smith')
        sue = Person('Sue Jones', job='dev', pay=100000)
        print(bob.name, bob.pay)
        print(sue.name, sue.pay)
```

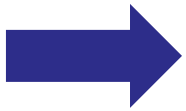
```
% python person.py
'Bob Smith', 0
'Sue Jones', 100000

% python
>>> from person import Person
>>> p = Person('James')
>>> p.name
'James'
```

각 클래스 모듈을 테스트할 수 있는 코드를 포함하는 것은
좋은 코딩 습관!

메서드 (Methods) 의 코딩

```
>>> from person import Person

>>> bob = Person('Bob Smith')
>>> sue = Person('Sue Jones', job='dev', pay=100000)

>>> bob.name.split()[-1]
'Smith'

>>> sue.pay *= 1.1
>>> print(sue.pay)
110000.0

>>> sue.pay
110000.00000000001
```

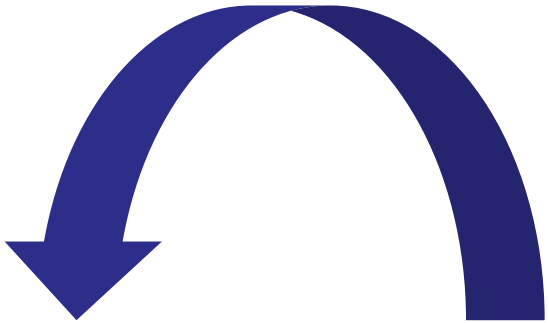
이렇게 할 수도 있지만,



```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(0.1)
    print(sue.pay)
```

연산자 오버로딩 (Operator Overloading)

```
>>> from person import Person
>>> sue = Person('Sue Jones', job='dev', pay=100000)
>>> print(sue)
<person.Person instance at 0x10079ab48>
```



```
% python person.py
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
('Smith', 'Jones')
110000
```

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(0.1)
    print(sue.pay)
```

하위 클래스 - Subclassing

하위 클래스 (subclass) 는 상위 클래스 (superclass) 의 속성과 메서드를 물려받지만,
메서드를 새롭게 작성함으로써 그와는 다른 동작을 하도록 개성화할 수 있다.

manager.py

```
from person import Person

class Manager(Person):
        def giveRaise(self, percent, bonus=.10):
                Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
        tom = Manager('Tom Jones', 'mgr', 50000)
        tom.giveRaise(.10)
        print(tom.lastName())
        print(tom)
```

```
% python manager.py
Jones
[Person: Tom Jones, 60000]
```

OOP 가 힘을 발휘할 때

```
from person import Person
from manager import Manager

class Department:
        def __init__(self, *args):
                self.members = list(args)
        def addMember(self, person):
                self.members.append(person)
        def giveRaises(self, percent):
                for person in self.members:
                        person.giveRaise(percent)
        def showAll(self):
                for person in self.members:
                        print(person)

if __name__ == '__main__':
        bob = Person('Bob Smith')
        sue = Person('Sue Jones', job='dev', pay=100000)
        tom = Manager('Tom Jones', 'mgr', 50000)
        development = Department(bob, sue)
        development.addMember(tom)
        development.giveRaises(.10)
        development.showAll()
```

```
% python department.py
[Person: Bob Smith, 0]
[Person: Sue Jones, 110000]
[Person: Tom Jones, 60000]
```

__init__(), __str__() 는 상위 클래스의 메서드가 실행
giveRaise() 는 Person 과 Manager 각각의 메서드가 실행

클래스 속성들 (Class Attributes)

```
>>> from person import Person
>>> bob = Person('Bob Smith')
>>> print(bob)
[Person: Bob Smith, 0]

>>> bob.__class__
<class person.Person at 0x1007834c8>

>>> bob.__class__.__name__
'Person'

>>> list(bob.__dict__.keys())
['pay', 'job', 'name']

>>> dir(bob)
['__doc__', '__init__', '__module__', '__str__', 'giveRaise',
'job', 'lastName', 'name', 'pay']
```

```
>>> for key in bob.__dict__:
...     print('%s => %s' % (key, bob.__dict__[key]))
...
pay => 0
job => None
name => Bob Smith

>>> for key in bob.__dict__:
...     print('%s => %s' % (key, getattr(bob, key)))
...
pay => 0
job => None
name => Bob Smith
```

클래스와 공유되는 속성들

```
>>> class SharedData:
...     spam = 42
...
>>> x = SharedData()
>>> y = SharedData()

>>> x.spam, y.spam
(42, 42)
```

```
>>> SharedData.spam = 99
>>> x.spam, y.spam, SharedData.spam
(99, 99, 99)

>>> x.spam = 88
>>> x.spam, y.spam, SharedData.spam
(88, 99, 99)
```

클래스의 데이터 객체를 공유하고 있음

인스턴스의 데이터 객체에 대한 대입문은
다른 인스턴스나 클래스에 영향을 미치지 않음

그러나!

```
>>> SharedData.spam = 100
>>> x.spam, y.spam, SharedData.spam
(88, 100, 100)
```

__getitem__()

클래스 내에 정의된 (또는 상속된) __getitem__ 메서드는

인스턴스에 대한 인덱싱 (indexing) 연산이 행해질 때 자동으로 호출됨

```
>>> class Indexer:
...     def __getitem__(self, index):
...             return index ** 2
...
>>> X = Indexer()

>>> X[2]
4

>>> for i in range(5):
...     print(X[i], end=' ')
0 1 4 9 16
```

__iter__() 와 __next__()

range() 처럼 동작하는 “iterator” 를 만들어낼 수 있음

- __iter__() 메서드는 __next__() 메서드가 정의된 클래스의 객체를 리턴
- __next__() 메서드는 이번 반복에서 계산될 값을 리턴
- 반복이 끝날 때에는 StopIteration 예외를 통하여 알림

iters.py

```
class Squares:
    def __init__(self, start, stop):
        self.value = start - 1
        self.stop = stop
    def __iter__(self):
        return self
    def __next__(self):
        if self.value == self.stop:
            raise StopIteration
        self.value += 1
        return self.value ** 2
```

```
>>> from iters import Squares
>>> for i in Squares(1, 5):
...     print(i, end=' ')
1 4 9 16 25
```

Quiz

- 클래스와 메서드 등, OOP 관련한 Python 프로그래밍을 배우는 이유는 무엇인가?
- 왜 클래스의 데이터 객체를 조작하는 기능은 메서드로 구현하는 것이 좋은가?
- 하위 클래스의 생성자에서, 공통된 기능에 대해서는 왜 상위 클래스의 생성자를 호출하는 것이 같은 코드를 반복하는 것보다 나은가?

Exercise

- 다음의 명세를 만족하는 상위 클래스와 하위 클래스를 만들어 보자.
  - 상위 클래스: Animal
  - 하위 클래스: Dog, Cat
  - Animal 의 속성:
    - feet (발의 개수)
    - sound (울음소리)
  - Animal 이 가지는 메서드:
    - 생성자 (인자로 동물의 이름을 받는다)
    - 문자열 포맷 (__str__())
    - walk() - 동일한 발자국 소리를 낸다.
    - talk() - 각각이 가지는 울음 소리를 낸다.

Q & A