기초빅데이터프로그래밍

객체지향 프로그래밍

서강대학교
SOGANG UNIVERSITY

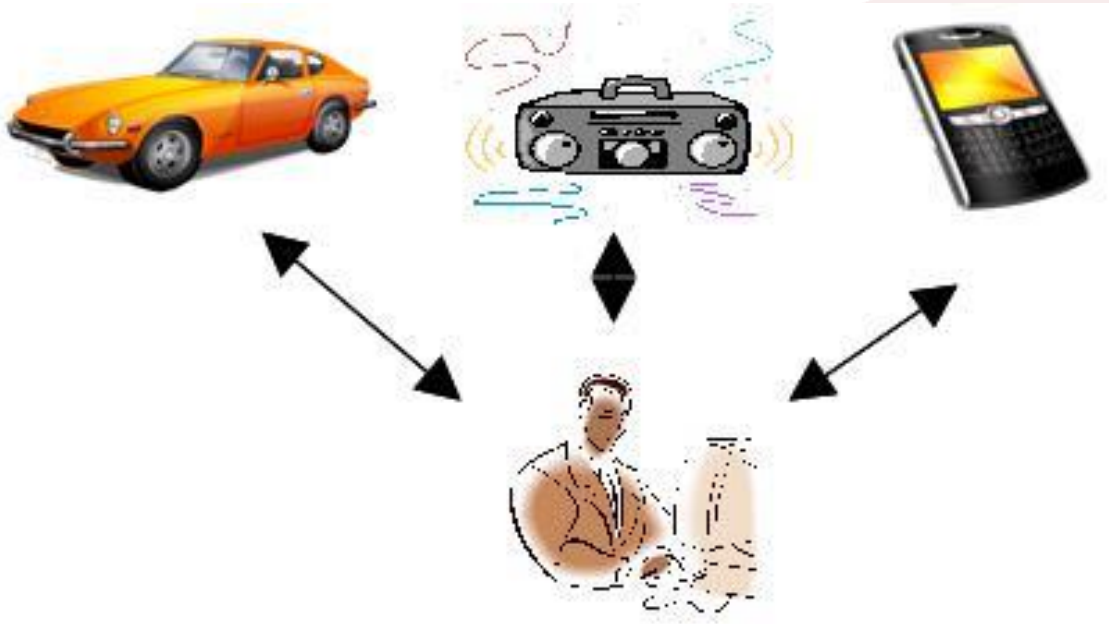
목차

목차

1 객체지향 프로그래밍이란?

- 객체지향 프로그래밍

  프로그램을 명령어들의 목록이 아니라, 객체들의 모임으로서, 객체와 객체와의 상호 관계를 중심으로 작성하자는 패러다임.



2 객체와 클래스

- **객체(object)**
  - **속성(Attribute)**과 **행위(Action)**를 가지고 있으며 이름을 붙일 수 있는 물체
    - 예: 자동차, 학생, 스마트 폰, 모니터, 키보드, 형광등, 책, 리모컨 등
- **클래스(class)**
  - 객체를 만들기 위한 설계도(blueprint)
- **인스턴스(instance):** 설계도에 따라 실제로 구현된 것
  - 인스턴스(instance)화 == 실체화 == 메모리에 구축한다
  - 클래스를 실체화 혹은 인스턴스화 시킨 것이 객체이다.

예로, class Programmer():

    pass

kim = Programmer()

kim이라는 객체는 Programmer라는 클래스의 인스턴스이다.

객체 생성예

```
student = {'name': '홍길동', 'year': 2, 'class': 3, 'student_id': 35}

print(student['name'], student['year'], student['class'], student['student_id'])
```

```
홍길동 2 3 35
```

```
class Student(object):
    def __init__(self, name, year, class_num, student_id):
        self.name = name
        self.year = year
        self.class_num = class_num
        self.student_id = student_id

    def introduce_myself(self):
        print(self.name, self.year, self.class_num, self.student_id)

student = Student('홍길동', 2, 3, 35)
student.introduce_myself()
```

```
홍길동 2 3 35
```

필요한 데이터에 접근하는 또 다른 방법

```
student = {'name': '홍길동', 'year': 2, 'class': 3, 'student_id': 35}

print(student['name'], student['year'], student['class'], student['student_id'])
```

```
홍길동 2 3 35
```

```
%%writefile stdu.py

name = '홍길동'
year = 2
class_num = 3
student_id = 35
```

```
Overwriting stdu.py
```

```
import stdu
print(stdu.name, stdu.year, stdu.class_num, stdu.student_id)
```

```
홍길동 2 3 35
```

Class

- 클래스 정의 문법

```
class 클래스_이름:
    클래스_본체
```

- Car 클래스의 속성과 행위

구분	코드	설명
클래스 이름	Car	자동차 클래스 이름
속성	_speed	속도
기능	get_speed	속도 가져오기
	start	출발
	accelerate	가속
	stop	정지

```
# car_example1.py

class Car:
    def __init__(self):
        self._speed = 0
        print("자동차가 생성되었습니다.")

    def get_speed(self):
        return self._speed

    def start(self):
        self._speed = 20
        print("자동차가 출발합니다.")

    def accelerate(self):
        self._speed = self._speed + 30
        print("자동차가 가속을 합니다.")

    def stop(self):
        self._speed = 0
        print("자동차가 정지합니다.")
```

```
if __name__ == "__main__":
    my_car = Car( )                                  # 1
    my_car.start()                                   # 2
    print("속도:", my_car.get_speed())               # 3
    my_car.accelerate()                              # 4
    print("속도:", my_car.get_speed())
    my_car.stop()                                    # 5
    print("속도:", my_car.get_speed())
```

#1: **Car 객체를 생성**, my_car라는 이름을 부여한다.
#2: my_car가 가진 start 메서드를 호출한다.
#3: get_speed 메서드를 호출하여 _speed 값을 반환 받아 화면에 출력
#4: accelerate 메서드를 호출하여 _speed값을 30만큼 증가
#5: stop 메서드를 호출하여 _speed 값을 0으로 설정

```
class Car:
    def __init__(self):
        self._speed = 0
        print("자동차가 생성되었습니다.")

    def get_speed(self):
        return self._speed

    def start(self):
        self._speed = 20
        print("자동차가 출발합니다.")

    def accelerate(self):
        self._speed = self._speed + 30
        print("자동차가 가속을 합니다.")

    def stop(self):
        self._speed = 0
        print("자동차가 정지합니다.")

if __name__ == "__main__":
    my_car = Car( )
    my_car.start()
    print("속도:", my_car.get_speed())
    my_car.accelerate()
    print("속도:", my_car.get_speed())
    my_car.stop()
    print("속도:", my_car.get_speed())
```

```
자동차가 생성되었습니다.
자동차가 출발합니다.
속도: 20
자동차가 가속을 합니다.
속도: 50
자동차가 정지합니다.
속도: 0
```

클래스명은 PEP 8 Coding Convention에 가이드된 대로 각 단어의 첫 문자를 대문자로 하는 **CapWords 방식**으로 명명한다.

3 self

- self는 객체 자신을 가리키는 특별한 변수이다.
- 현재 인스턴스 객체를 가리키는 것으로, C++나 자바의 this 키워드와 동일하지만, 예약어로 지정하지는 않았다.
- 객체 메서드에는 반드시 첫 번째 인자로 self가 와야 한다.

```
class Person:
    Name = "Default Name 홍길동"
    def Print(self):
        print("My Name is {0}".format(self.Name))
```

```
p1 = Person()
```

```
p1.Print()
```

```
My Name is Default Name 홍길동
```

```
p1.Name = "서경희"
```

```
p1.Print()    # bound method call
```

```
My Name is 서경희
```

```
Person.Print(p1)   # unbound method call
```

```
My Name is 서경희
```

Bound method call

```
class Foo:
    def func1():
        print("Function 1")
    def func2(self):
        print("function 2")
```

```
f=Foo()
f.func2()
```

```
function 2
```

```
f.func1()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-b3c056b64a8f> in <module>()
----> 1 f.func1()

TypeError: func1() takes 0 positional arguments but 1 was given
```

func1은 인자가 없는데 1개 받았다고 함. 즉, 첫 번째 인자로 항상 인스턴스가 전달되기 때문에 발생한 문제이다.

```
In [7]: class Foo:
            def func1():
                print("Function 1")
            def func2(self):
                print(id(self))
                print("function 2")
```

```
In [8]: f=Foo()
        id(f)
```

```
Out[8]: 2113307554592
```

```
In [9]: f.func2()
```

```
2113307554592
function 2
```

```
In [10]: f2=Foo()
         id(f2)
```

```
Out[10]: 2113307555712
```

```
In [11]: f2.func2()
```

```
2113307555712
function 2
```

f의 바인딩주소는 인스턴스의 주소값

```
In [12]: Foo.func1()

Function 1

In [13]: Foo.func2()

---------------------------------------------------------------------
----------
TypeError                                 Traceback (most recent
call last)
<ipython-input-13-1213fa831820> in <module>()
----> 1 Foo.func2()

TypeError: func2() missing 1 required positional argument: 'self'

In [14]: f3=Foo()
         id(f3)

Out[14]: 2113307556776

In [15]: Foo.func2(f3)

2113307556776
function 2

In [16]: f3.func2()

2113307556776
function 2
```

- 파이썬 클래스 자체가 하나의 네임스페이스이므로 인스턴스의 생성과 무관하게 클래스내의 메서드 호출가능(클래스와 인스턴스의 네임스페이스가 분리)
- Foo.func1: ㅇㅋ
- Foo.func2는 인자를 전달하지 않아 에러발생

- 인스턴스를 생성해서 그 인스턴스 f3 전달해줌

왜 OOP인가?

각각의 결과값을 유지해야 하는 2개의 계산기가 필요한 경우 함수 두 개로 해결했지만, 계산기가 5개, 10개로 **점점 더 많이 필요해진다면** 어떻게 해야 할 것인가? **그때마다 전역변수와 함수를 추가할 것인가?**

In [1]:
```
result1 = 0
result2 = 0

def adder1(num):
    global result1
    result1 += num
    return result1

def adder2(num):
    global result2
    result2 += num
    return result2

print(adder1(3))
print(adder1(4))
print(adder2(3))
print(adder2(7))
```

```
3
7
3
10
```

[2]:
```
class Calculator:
    def __init__(self):
        self.result = 0

    def adder(self, num):
        self.result += num
        return self.result

cal1 = Calculator()
cal2 = Calculator()

print(cal1.adder(3))
print(cal1.adder(4))
print(cal2.adder(3))
print(cal2.adder(7))
```

```
3
7
3
10
```

계산기의 개수가 늘어나더라도 객체를 생성하기만 하면 되기 때문에 함수를 사용하는 경우와 달리 매우 간단해진다.

실습: 사칙연산하는 클래스 만들기

다음과 같이 사칙연산을 가능하게 하는 FourCal이라는 클래스를 만드시오.

```
In [4]: a=FourCal(4,2)
        b=FourCal(3,7)
        a.sum()
Out[4]: 6

In [3]: a.mul()
Out[3]: 8

In [4]: a.sub()
Out[4]: 2

In [5]: a.div()
Out[5]: 2.0

In [6]: b.sum()
Out[6]: 10

In [7]: b.mul()
Out[7]: 21

In [6]: b.sub()
Out[6]: -4

In [7]: b.div()
Out[7]: 0.4285714285714285
```

실습 예

In [2]:
```
class FourCal:
    def __init__(self, first, second):
        self.first = first
        self.second = second
    def sum(self):
        result = self.first + self.second
        return result
    def mul(self):
        result = self.first* self.second
        return result
    def sub(self):
        result = self.first - self.second
        return result
    def div(self):
        result = self.first / self.second
        return result
```

In [4]:
```
a=FourCal(4,2)
b=FourCal(3,7)
a.sum()
```

Out[4]: 6

In [3]:
```
a.mul()
```

Out[3]: 8

Class Members

- Class Members
  - **method**
  - **Property @**
  - **class variable**
  - **instance variable**
  - **Initializer __init__**
  - **Destructor __del__**

- 데이터를 표현하는 **field**와 행위를 표현하는 **method**로 구분하며, 파이썬에서는 이러한 field와 method 모두 그 객체의 attribute라고도 한다.
- 새로운 attribute를 동적으로 추가할 수 있으며, 메서드도 일종의 메서드 객체로 취급하여 attribute에 포함한다.

Initializer

- **__init__( )**
- 클래스로부터 새 객체가 생성될 때 마다 실행되는 특별한 메서드 (magic method)
- 인스턴스 변수의 초기화, 객체의 초기 상태를 만듦
- Python에서 클래스 생성자(Constructor)는 실제 런타임 엔진 내부에서 실행되는데, 이 생성자 실행 도중 클래스 안에 Initializer가 있는지 체크하여 만약 있으면 Initializer를 호출하여 객체의 변수 등을 초기화한다.

Destructor(소멸자)

- __del__
- 클래스로부터 객체가 소멸할 때 호출되는 특별한 메서드
- 객체의 reference counter(참조 카운터)가 0 이 되면 자동 호출
- 리소스 해제 등의 종료작업 수행

생성자와 소멸자의 예

```
class IceCream:
    def __init__(self, name, price):
        self.name = name
        self.price = price
        print(name + "의 가격은 "+ str(price) + "원 입니다.")
    def __del__(self):
        print(self.name + " 객체가 사라집니다")
```

```
obIce = IceCream("누가바", 800)
```

```
누가바의 가격은 800원 입니다.
```

```
ob3Ice = IceCream("월드콘", 1000)
```

```
월드콘의 가격은 1000원 입니다.
```

```
del ob3Ice
```

```
월드콘 객체가 사라집니다
```

소멸자의 예

```
class IceCream:
    def __init__(self, name, price):
        self.name = name
        self.price = price
        print(self.name + "의 가격은 "+ str(self.price) + "원 입니다.")
    def __del__(self):
        print(self.name + " 객체가 사라집니다")
```

```
obIce = IceCream("누가바", 800) #rc of obIce =1
```

누가바의 가격은 800원 입니다.

```
ob2Ice = IceCream("월드콘", 1000) #rc of ob2Ice =1
```

월드콘의 가격은 1000원 입니다.

```
obIce_copy = obIce  # rc of obIce =2
```

```
del obIce #rc of obIce =1
```

```
del obIce_copy #rc of obIce =0, 소멸자 호출
```

누가바 객체가 사라집니다

객체의 reference counter가 1 이상이면 del 구문을 사용해도 destructor가 호출되지 않는다

```python
class Employee(object):
    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay
        self.email = first.lower()+'.'+last.lower()+'@sogang.ac.kr'

    def full_name(self):
        print(self.first + ' '+ self.last)

emp_1 = Employee('Jiho', 'Lee', 50000)
emp_2 = Employee('Minjung', 'Kim', 60000)
print(emp_1.email)
print(emp_2.email)
# emp_1의 풀네임 출력
emp_1.full_name()
```

```
jiho.lee@sogang.ac.kr
minjung.kim@sogang.ac.kr
Jiho Lee
```

```
class Employee(object):
    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay
        self.email = first.lower()+'.'+last.lower()+'@sogang.com'

    def __del__(self):
        print(self.last + " 퇴사했습니다")

    def full_name(self):
        print(self.first + ' '+ self.last)
```

```
emp_1 = Employee('Jiho', 'Lee', 50000)
emp_2 = Employee('Minjung', 'Kim', 60000)
print(emp_1.email)
print(emp_2.email)
emp_1.full_name()
```

```
jiho.lee@sogang.com
minjung.kim@sogang.com
Jiho Lee
```

```
del emp_1
```

```
Lee 퇴사했습니다
```

클래스 변수와 인스턴스 변수

클래스 변수가 하나의 클래스에 하나만 존재하는 반면, 인스턴스 변수는 각 객체 인스턴스마다 별도로 존재한다.

인스턴스 변수: 클래스 정의에서 메서드 안에서 사용되면서 "**self.변수명**"처럼 사용되며, 이는 **각 객체 별로 서로 다른 값을 갖는 변수이다**.

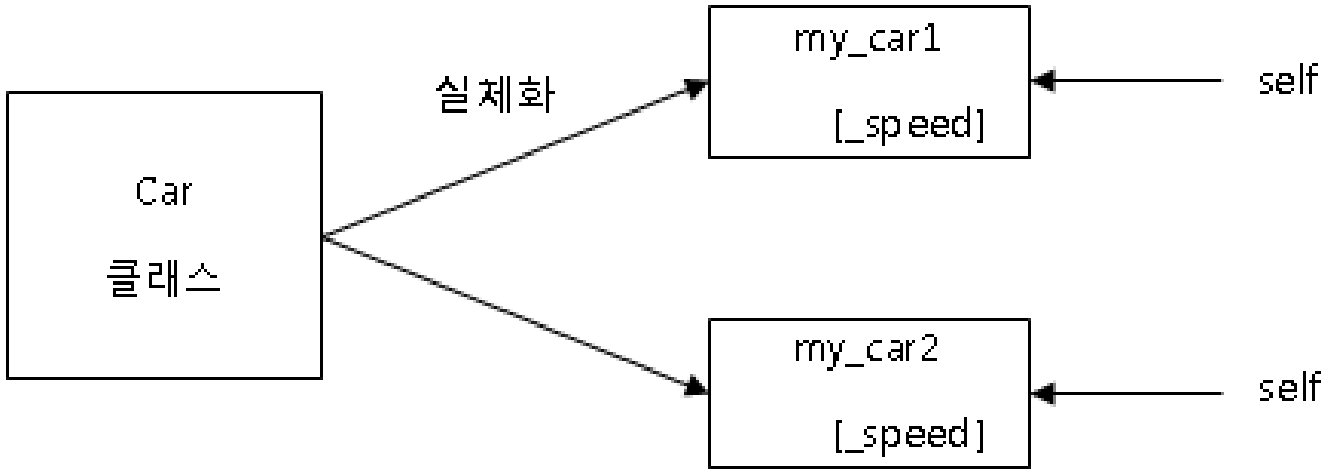
```
In [6]: class Account:
            num_accounts = 0 #클래스 변수
            def __init__(self, name):
                self.name = name     #인스턴스 변수
                Account.num_accounts += 1
            def __del__(self):
                Account.num_accounts -= 1

In [8]: kim = Account("kim")
        kim.name
Out[8]: 'kim'

In [9]: kim.num_accounts
Out[9]: 1

In [10]: Account.num_accounts
Out[10]: 1

In [11]: lee = Account("lee")

In [12]: lee.name
Out[12]: 'lee'

In [13]: lee.num_accounts
Out[13]: 2

In [14]: Account.num_accounts
Out[14]: 2
```

```
# car_example2.py

class Car:
    def __init__(self):
        self._speed = 0

    def get_speed(self):
        return self._speed

    def start(self):
        self._speed = 20

    def accelerate(self):
        self._speed = self._speed + 30

    def stop(self):
        self._speed = 0
```

```
if __name__ == "__main__":
    my_car1 = Car( )
    my_car2 = Car( )
    my_car1.start( )
    my_car2.start( )
    my_car1.accelerate( )
    my_car2.accelerate( )
    my_car2.accelerate( )
    print("첫번째 자동차 속도:", my_car1.get_speed( ))
    print("두번째 자동차 속도:", my_car2.get_speed( ))
    my_car1.stop( )
    my_car2.stop( )
```

```
class Car:
    def __init__(self):
        self._speed = 0

    def get_speed(self):
        return self._speed

    def start(self):
        self._speed = 20

    def accelerate(self):
        self._speed = self._speed + 30

    def stop(self):
        self._speed = 0

if __name__ == "__main__":
    my_car1 = Car( )
    my_car2 = Car( )
    my_car1.start( )
    my_car2.start( )
    my_car1.accelerate( )
    my_car2.accelerate( )
    my_car2.accelerate( )
    print("첫번째 자동차 속도:", my_car1.get_speed( ))
    print("두번째 자동차 속도:", my_car2.get_speed( ))
    my_car1.stop( )
    my_car2.stop( )
```

```
첫번째 자동차 속도: 50
두번째 자동차 속도: 80
```



- **self를 이용해서 객체 자신이 가진 속성을 변화시키기 때문에 객체의 속성 값이 정확하게 관리된다.**

구분	코드	설명
클래스 이름	**Radio**	라디오 클래스 이름
속성		
기능	turn_on turn_off	라디오를 켠다 라디오를 끈다

구분	코드	설명
클래스 이름	**Car**	자동차 클래스 이름
속성	_speed _radio	속도 라디오
기능	get_speed start accelerate stop turn_on_radio turn_off_radio	속도 가져오기 출발 가속 정지 라디오를 켠다 라디오를 끈다

```
# car_with_radio.py

class Radio:
    def __init__(self):
        print("라디오가 생성되었습니다.")

    def turn_on(self):
        print("라디오를 켭니다.")

    def turn_off(self):
        print("라디오를 끕니다.")
```

```
class Car:
    def __init__(self):
        self._speed = 0
        print("자동차가 생성되었습니다.")
        self._radio = Radio( )

    def get_speed(self):
        return self._speed

    def start(self):
        self._speed = 20
        print("자동차가 출발합니다.")

    def accelerate(self):
        self._speed = self._speed + 30
        print("자동차가 가속을 합니다.")

    def stop(self):
        self._speed = 0
        print("자동차가 정지합니다.")

    def turn_on_radio(self):
        self._radio.turn_on( )

    def turn_off_radio(self):
        self._radio.turn_off( )
```

```
if __name__ == "__main__":
    my_car = Car( )
    my_car.start()
    my_car.turn_on_radio( ) # 1
    my_car.accelerate()
    my_car.turn_off_radio( ) # 2
    my_car.stop()
```

```
class Radio:
    def __init__(self):
        print("라디오가 생성되었습니다.")

    def turn_on(self):
        print("라디오를 켭니다.")

    def turn_off(self):
        print("라디오를 끕니다.")

class Car:
    def __init__(self):
        self._speed = 0
        print("자동차가 생성되었습니다.")
        self._radio = Radio()

    def get_speed(self):
        return self._speed

    def start(self):
        self._speed = 20
        print("자동차가 출발합니다.")

    def accelerate(self):
        self._speed = self._speed + 30
        print("자동차가 가속을 합니다.")

    def stop(self):
        self._speed = 0
        print("자동차가 정지합니다.")

    def turn_on_radio(self):
        self._radio.turn_on( )

    def turn_off_radio(self):
        self._radio.turn_off( )
```

```
if __name__ == "__main__":
    my_car = Car( )
    my_car.start()
    print("속도:", my_car.get_speed())
    my_car.turn_on_radio()
    my_car.accelerate()
    print("속도:", my_car.get_speed())
    my_car.turn_off_radio()
    my_car.stop()
    print("속도:", my_car.get_speed())
```

```
자동차가 생성되었습니다.
라디오가 생성되었습니다.
자동차가 출발합니다.
속도: 20
라디오를 켭니다.
자동차가 가속을 합니다.
속도: 50
라디오를 끕니다.
자동차가 정지합니다.
속도: 0
```

4 모듈과 클래스

- 함수와 클래스를 파이썬 인터프리터에서 바로 정의하고 사용하는 것이 가능하다.
- 그러나 인터프리터에서 빠져 나오면 사라지므로, 파일 내에 구축하는 것이 좋다.

```
>>> def change_km_to_mile(km):
...         return km * 0.621371
...
>>> class Car:
...         def __init__(self):
...             self._speed = 0
...         def get_speed(self):
...             return self._speed
...         def start(self):
...             self._speed = 20
...         def accelerate(self):
...             self._speed = self._speed + 20
...         def stop(self):
...             self._speed = 0
...
>>> my_car = Car( )
>>> my_car.start( )
>>> speed_km = my_car.get_speed( )
>>> print("현재 속도:", speed_km,"km")
현재 속도: 20 km
>>> speed_mile = change_km_to_mile(speed_km)
>>> print("현재 속도:", speed_mile, "mile")
현재 속도: 12.42742 mile
>>> exit()
```

vehicle.py

```
def change_km_to_mile(km):
    return km * 0.621371

def change_mile_to_km(mile):
    return mile * 1.609344

class Car:
    def __init__(self):
        self._speed = 0

    def get_speed(self):
        return self._speed

    def start(self):
         self._speed = 20

    def accelerate(self):
        self._speed = self._speed + 30

    def stop(self):
        self._speed = 0
```

```
class Truck:
    def __init__(self):
        self._speed = 0

    def get_speed(self):
        return self._speed

    def start(self):
        self._speed = 10

    def accelerate(self):
        self._speed = self._speed + 20

    def stop(self):
        self._speed = 0
```

```
if __name__ == "__main__":
    my_car = Car( )
    my_car.start( )
    my_car.accelerate( )
    print("속도:", my_car.get_speed( ))
    my_car.stop( )
```

vehicle.py

서강대학교 SOGANG UNIVERSITY

```
def change_km_to_mile(km):
    return km * 0.621371

def change_mile_to_km(mile):
    return mile * 1.609344

class Car:
    def __init__(self):
        self._speed = 0

    def get_speed(self):
        return self._speed

    def start(self):
         self._speed = 20

    def accelerate(self):
        self._speed = self._speed + 30

    def stop(self):
        self._speed = 0
```

```
class Truck:
    def __init__(self):
        self._speed = 0

    def get_speed(self):
        return self._speed

    def start(self):
        self._speed = 10

    def accelerate(self):
        self._speed = self._speed + 20

    def stop(self):
        self._speed = 0

if __name__ == "__main__":
    my_car = Car( )
    my_car.start( )
    my_car.accelerate( )
    print("속도:", my_car.get_speed( ))
    my_car.stop( )
```

```
속도: 50
```

모듈

- 여러 코드를 한데 묶어 다른 곳에서 **재사용할 수 있는 코드 모음**
- 파이썬 표준 라이브러리 URL
  - https://docs.python.org/3.8/library/index.html
- **모듈 이름을 name space**로 이용하여 관리

  예: import math ⇒ math라는 이름공간이 생성되며, math.attribute_name형식으로 모듈의 함수나 attribute이용
- 모듈 만들기는 **파일을 만든다는** 뜻이며, 일반적으로 **<모듈이름>.py**로 저장

  만든 모듈을 파이썬 표준 library directory에 저장하면 됨
- 파이썬에서는 import 구문을 어디서나 사용가능
- 모듈을 import하면, sys.path에 등록된 디렉토리에서 모듈을 찾는데, 모듈의 바이트 코드(.pyc file)가 있으면, 이 코드를 바로 임포트하고, 없으면 해당 모듈의 코드를 바이트 코드로 만든다.
- 바이트 코드는 내용변경이 없는 한 새로 만들어지지 않음.
- 모듈은 메모리에 한번만 로딩되고, 코드도 한번만 실행됨

import module ☞ dot (.)연산자 필요

```
# vehicle_example.py

import vehicle      # vehicle 모듈 import

if __name__ == "__main__":
    my_car = vehicle.Car( )
    my_car.start( )
    my_car.accelerate( )
    speed_mile = vehicle.change_km_to_mile(my_car.get_speed( ))
    print("속도:", speed_mile, "mile")
    my_car.stop( )
```

from module import * ☞ dot 연산자 필요 없음

```
# vehicle_example1.py

from vehicle import *          # vehicle 모듈 import

if __name__ == "__main__":
    my_car = Car( )
    my_car.start( )
    my_car.accelerate( )
    speed_mile = change_km_to_mile(my_car.get_speed( ))
    print("속도:", speed_mile, "mile")
    my_car.stop( )
```

```
%%writefile vehicle.py

def change_km_to_mile(km):
    return km * 0.621371

def change_mile_to_km(mile):
    return mile * 1.609344

class Car:
    def __init__(self):
        self._speed = 0

    def get_speed(self):
        return self._speed

    def start(self):
         self._speed = 20

    def accelerate(self):
        self._speed = self._speed + 30

    def stop(self):
        self._speed = 0

class Truck:
    def __init__(self):
        self._speed = 0

    def get_speed(self):
        return self._speed

    def start(self):
        self._speed = 10

    def accelerate(self):
        self._speed = self._speed + 20

    def stop(self):
        self._speed = 0

if __name__ == "__main__":
    my_car = Car( )
    my_car.start( )
    my_car.accelerate( )
    print("속도:", my_car.get_speed( ))
    my_car.stop( )
```

```
Writing vehicle.py
```

```
%run vehicle.py
```

```
속도: 50
```

```
from vehicle import *          # vehicle 모듈 import

if __name__ == "__main__":
    my_car = Car( )
    my_car.start( )
    my_car.accelerate( )
    speed_mile = change_km_to_mile(my_car.get_speed( ))
    print("속도:", speed_mile, "mile")
    my_car.stop( )
```

```
속도: 31.068550000000002 mile
```

5 클래스와 데이터 타입

2.X에서는 <class 'int'> 라고 표시되었었는데...

- **파이썬에서는 모든 것이 객체이다.**

```
>>> type(1)
<class 'int'>
>>> type(3.14)
<class 'float'>
>>> isinstance(1, int)
True
>>> type("Hello World")
<class 'str'>
>>> isinstance("Hello World", str)
>>> type([1,2,3,"안녕 파이썬!"])
<class 'list'>
>>> isinstance([1, 2, 3, "안녕 파이썬!"], list)
True
>>> type({"line":1, "rectangle":2, "triangle": 3})
<class 'dict'>
>>> isinstance({"line":1, "rectangle":2, "triangle":3}, dict)
True
```

```
type(1)
int

type(3.14)
float

isinstance(1, int)
True

type("Hello world")
str

isinstance("Hello world", str)
True
```

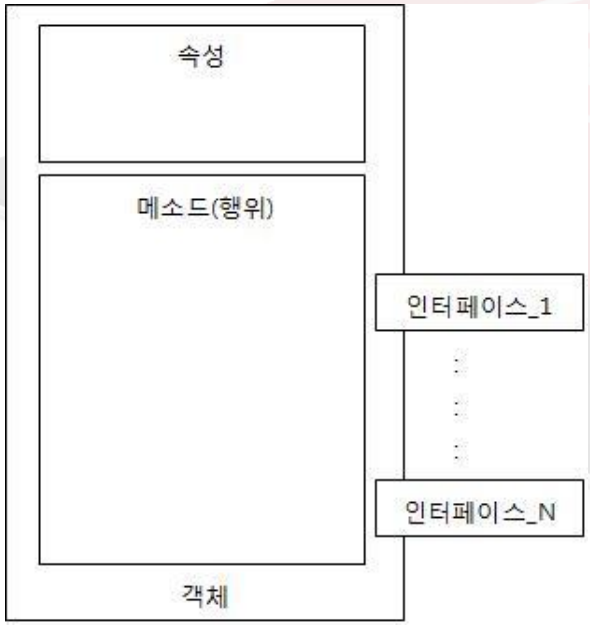
- **사용자 정의 클래스**

```
>>> from vehicle import *
>>> my_car = Car( )
>>> type(my_car)
<class 'Car'>
>>> isinstance(my_car, Car)
True
>>> my_truck = Truck( )
>>> type(my_truck)
<class 'Truck'>
>>> isinstance(my_truck, Truck)
True
```

```
In [8]:  from vehicle import *

In [9]:  my_car = Car()

In [10]: type(my_car)
Out[10]: vehicle.Car

In [11]: isinstance(my_car, Car)
Out[11]: True

In [13]: my_truck = Truck()
         type(my_truck)
Out[13]: vehicle.Truck

In [15]: isinstance(my_truck, Truck)
Out[15]: True
```

6 캡슐화와 접근 지정

- **캡슐화(encapsulation)**는 객체의 외부에서 객체 내부의 속성을 직접적으로 접근, 속성을 읽거나 변경시키지 못하게 하는 것을 말한다.
- 객체의 속성을 접근, 속성을 읽거나 변경하고 싶으면 반드시 **허락된 인터페이스를 통해서**만이 가능하다.
  - 객체가 외부로 **공개한 메서드를 뜻**한다.



Access Modifiers

- C++나 Java와 같은 OOP 언어에서는 객체의 속성이나 메서드에 접근을 제어하는 접근 지정자 (public, private, protected)가 제공된다.
- 파이썬에서는 이런 접근 지정자가 없다.
- 파이썬 클래스에서는 기본적으로 모든 멤버가 public이라고 할 수 있다.
- 파이썬에서의 접근 지정은 이름 규칙을 통해 처리되며, 프로그래머가 접근지정에 대한 책임을 지게 한다.
- 파이썬에서는 접근 지정을 구분하기 위해 밑줄(_)을 사용한다.
  - **밑줄이 없으면 :** 공개 모드로서 객체 외부에서 접근이 가능하다.
  - **밑줄이 하나이면 :** 객체 외부에서 접근해서는 안 된다.
    - 코딩 관례상 내부적으로만 사용되는 변수 또는 메서드에 사용
    - 그러나 public이므로 필요하면 외부에서 엑세스가능
  - **밑줄이 두 개이면 :** private 모드로서 객체 외부뿐만 아니라 상속받은 객체에서도 접근이 안 된다.

```
# access_modifiers1.py

class Car:
    def __init__(self):
        self.price = 2000
        self._speed = 0
        self.__color = "red"

if __name__ == "__main__":
    my_car = Car( )
    print(my_car.price) # 1
    print(my_car._speed) # 2
    print(my_car.__color) # 3
```

```
class Car:
    def __init__(self):
        self.price = 2000
        self._speed = 0
        self.__color = "red"

if __name__ == "__main__":
    my_car = Car( )
    print(my_car.price) # 1
    print(my_car._speed) # 2
    print(my_car.__color) # 3
```

```
2000
0
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent ca
<ipython-input-1-ad9551b8fdba> in <module>()
     11     print(my_car.price) # 1
     12     print(my_car._speed) # 2
---> 13     print(my_car.__color) # 3

AttributeError: 'Car' object has no attribute '__color'
```

1 public mode
2 접근 가능하지만 메서드 통해 접근권장
3 private mode

```
# access_modifiers2.py

class Car:
    def __init__(self):
        self._price = 0
        self._speed = 0
        self._color = None

    def get_price(self):
        return self._price

    def set_price(self, value):
        self._price = value

    def get_speed(self):
        return self._speed

    def set_speed(self, value):
        self._speed = value

    def get_color(self):
        return self._color

    def set_color(self, value):
        self._color = value
```

객체의 속성은 밑줄 하나로 시작하게 하고,
외부에서는 메서드를 통해 접근하도록 한다.

get/set method

```
if __name__ == "__main__":
    my_car = Car( )
    my_car.set_price(2000)
    my_car.set_speed(20)
    my_car.set_color("red")
    print("가격:", my_car.get_price( ))
    print("속도:", my_car.get_speed( ))
    print("색상:", my_car.get_color( ))
```

```
class Car:
    def __init__(self):
        self._price = 0
        self._speed = 0
        self._color = None

    def get_price(self):
        return self._price

    def set_price(self, value):
        self._price = value

    def get_speed(self):
        return self._speed

    def set_speed(self, value):
        self._speed = value

    def get_color(self):
        return self._color

    def set_color(self, value):
        self._color = value

if __name__ == "__main__":
    my_car = Car( )
    my_car.set_price(2000)
    my_car.set_speed(20)
    my_car.set_color("red")
    print("가격:", my_car.get_price( ))
    print("속도:", my_car.get_speed( ))
    print("색상:", my_car.get_color( ))
```

```
가격: 2000
속도: 20
색상: red
```

7 property 이용하기

- **get/set 메서드**
  - 단지 객체의 속성 값을 읽거나 설정하는데 사용되는 메서드
- **property**
  - 객체의 속성을 접근하는 것이 **마치 속성에 직접적으로 접근하는 것처럼 보이지만** 내부적으로는 메서드를 통해 접근하도록 하는 방식

- 문법

```
@property                          # 다음의 메서드_이름으로 접근 가능
def 메서드_이름(self):
    return self._메서드_이름
@메서드_이름.setter                 # 다음의 메서드_이름으로 설정 가능
def 메서드_이름(self, value):
    self._메서드_이름 = value
```

```
# car_property1.py

class Car:
    def __init__(self):
        self._price = 0
        self._speed = 0
        self._color = None

    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, value):
        self._price = value

    @property
    def speed(self):
        return self._speed

    @speed.setter
    def speed(self, value):
        self._speed = value

    @property
    def color(self):
        return self._color

    @color.setter
    def color(self, value):
        self._color = value
```

price, speed, color가 내부적으로 메서드이다.

```
if __name__ == "__main__":
    my_car = Car( )
    my_car.price = 2000
    my_car.speed = 20
    my_car.color = "red"
    print("가격:", my_car.price)
    print("속도:", my_car.speed)
    print("색상:", my_car.color)
```

객체의 속성을 접근하듯이 사용하지만, price, speed, color가 내부적으로 메서드이다.

```
class Car:
    def __init__(self):
        self._price = 0
        self._speed = 0
        self._color = None

    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, value):
        self._price = value

    @property
    def speed(self):
        return self._speed

    @speed.setter
    def speed(self, value):
        self._speed = value

    @property
    def color(self):
        return self._color

    @color.setter
    def color(self, value):
        self._color = value

if __name__ == "__main__":
    my_car = Car( )
    my_car.price = 2000
    my_car.speed = 20
    my_car.color = "red"
    print("가격:", my_car.price)
    print("속도:", my_car.speed)
    print("색상:", my_car.color)
```

```
가격: 2000
속도: 20
색상: red
```

property class 이용

- @ decorator : 전통적인 방식
- 파이썬 2.6 이후부터 property class가 제공됨

- property( ) 클래스의 문법

  property object을 만들어 돌려주는 built-in function

  property(fget=None, fset=None, fdel=None, doc=None) -> property attribute

  fget: get 함수를 설정하는 인자

  fset: set 함수를 설정하는 인자

  fdel: delete 함수를 설정하는 인자

  doc: 이 속성의 설명을 설정하는 인자

```
# car_property2.py

class Car:
    def __init__(self):
        self._price = 0
        self._speed = 0
        self._color = None

    def get_price(self):
        return self._price

    def set_price(self, value):
        self._price = value

    price = property(get_price, set_price)
```

```
    def get_speed(self):
        return self._speed

    def set_speed(self, value):
        self._speed = value

    speed = property(get_speed, set_speed)

    def get_color(self):
        return self._color

    def set_color(self, value):
        self._color = value

    color = property(get_color, set_color)
```

```
if __name__ == "__main__":
    my_car = Car( )
    my_car.price = 2000
    my_car.speed = 20
    my_car.color = "red"
    print("가격:", my_car.price)
    print("속도:", my_car.speed)
    print("색상:", my_car.color)
```

배정문에서의 my_car.price는 자동으로 set_price를 호출함

```
class Car:
    def __init__(self):
        self._price = 0
        self._speed = 0
        self._color = None

    def get_price(self):
        return self._price

    def set_price(self, value):
        self._price = value
    price = property(get_price, set_price)

    def get_speed(self):
        return self._speed

    def set_speed(self, value):
        self._speed = value
    speed = property(get_speed, set_speed)

    def get_color(self):
        return self._color

    def set_color(self, value):
        self._color = value
    color = property(get_color, set_color)

if __name__ == "__main__":
    my_car = Car( )
    my_car.price = 2000
    my_car.speed = 20
    my_car.color = "red"
    print("가격:", my_car.price)
    print("속도:", my_car.speed)
    print("색상:", my_car.color)
```

```
가격: 2000
속도: 20
색상: red
```

property() 와 @

```
class C(object):
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x
    def setx(self, value):
        self._x = value
    def delx(self):
        del self._x
    x = property(getx, setx, delx, "I'm

if __name__ == "__main__":
    my_c = C()
    my_c.x=30
    print("value of x", my_c.x)
```

```
value of x 30
```

```
class C(object):
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x

if __name__ == "__main__":
    my_c = C()
    my_c.x=30
    print("value of x", my_c.x)
```

```
value of x 30
```

Decorator

- any callable Python object that is used to modify a function, method or class definition
- A decorator is passed the original object being defined and returns a modified object, which is then bound to the name in the definition.
- pure syntactic sugar, using @ as the keyword:
- In Python prior to version 2.6, decorators apply to functions and methods, but not to classes.
- Class decorators are supported starting with Python 2.6.
- Python decorators add functionality to functions and methods at definition time, not at run time
- Canonical uses of function decorators are for creating class methods or static methods, adding function attributes, tracing, setting pre- and postconditions, and synchronization, but can be used for far more besides, including tail recursion elimination, memoization and even improving the writing of decorators.

Decorator 사용 예

```
def verbose(func):
    def new_func():
        print ("Begin", func.__name__)
        func()
        print ("End", func.__name__)
    return new_func

@verbose
def my_function():
    print ("hello, world.")

print ("Program start")
my_function()
```

```
Program start
Begin my_function
hello, world.
End my_function
```

Decorator는 함수, callable object이므로, 데코레이션하기 전에 해당 함수 또는 클래스를 정의해야한다.

```
def verbose(func):
    def new_func():
        print ("Begin", func.__name__)
        func()
        print ("End", func.__name__)
    return new_func

def my_function():
    print ("hello, world.")

my_function = verbose(my_function)
print ("Program start")
my_function()
```

```
Program start
Begin my_function
hello, world.
End my_function
```

함수로 구현한 예, 파이썬에서의 함수는 first class 함수이므로

Decorator 사용 예

1. **데코레이터로 사용할 함수**를 하나 정의한다. 이 함수는 데코레이터가 적용될 함수를 인자로 받는다.

2. 1의 함수는 내부에서 신규 함수를 정의한다. 신규함수는 기존 함수를 내부에서 호출하며, 그 앞/뒤로 다른 동작을 수행할 수 있다. 가능하면 가변 인자를 사용해 새 함수를 정의한다.

3. 이제 데코레이터를 적용할 함수를 정의하기 전에 @데코레이터 형식의 구문을 앞 줄에 미리 써준다.

```
def verbose(func):
    def new_func():
        print ("\nBegin", func.__name__)
        func()
        print ("End", func.__name__)
    return new_func

@verbose
def my_function():
    print ("hello, world.")

@verbose
def your_function():
    print ("hello, city.")

@verbose
def our_function():
    print ("hello, everybody.")
print ("Program start")
my_function()
your_function()
our_function()
```

```
Program start

Begin my_function
hello, world.
End my_function

Begin your_function
hello, city.
End your_function

Begin our_function
hello, everybody.
End our_function
```

Decorator 사용 예: 클래스 이용

```
class verbose:
    def __init__(self, f):
        print("Initializing Verbose.")
        self.func = f

    def __call__(self):
        print ("Begin", self.func.__name__)
        self.func()
        print ("End", self.func.__name__, "\n")

@verbose
def my_function():
    print ("hello, world.")

print("Program start\n")
my_function()
```

```
Initializing Verbose.
Program start

Begin my_function
hello, world.
End my_function
```

클래스로 decorator를 정의한 예

클래스를 함수처럼 호출되게 하려면 __call__() 멤버 함수를 정의하면 된다.

실습 1

- **인자를 갖는 함수에 대한 decorator를 만드시오.**

다음과 같은 출력을 보이도록
앞의 예제를 수정하시오.

```
print ("Program start")
my_function("Mickey");
my_function("Minnie")
my_function("Donald")
```

출력

```
Initializing Verbose.
Program start

Begin my_function
hello, Mickey!
End my_function

Begin my_function
hello, Minnie!
End my_function

Begin my_function
hello, Donald!
End my_function
```

실습 2

- 실행시간을 구하는 함수 checkTime(func)을 만들어서 decorator로 이용하여, 다음 두 함수, aFunc()와 bFunc에 대해 실행시간을 구하고, 실행시작 날짜와 시간을 다음과 같이 출력하시오. 즉, 아래와 같은 출력을 보이도록 함수 checkTime(func), aFunc(), bFunc(start, end)을 만드시오. 이때, checkTime은 인자가 없는 함수 aFunc도 인자가 있는 함수 bFunc도 처리할 수 있어야 합니다 .

- 실행 예

```
aFunc()
print("-----------------------")
bFunc(101,202)
```

```
[2020-04-23 17:14]
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 3
0 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 8
3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
실행시간은: 0.0029981136322021484
-----------------------
[2020-04-23 17:14]
101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120
121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140
141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160
161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180
181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200
201 202
실행시간은: 0.001998424530029297
```