빅데이터 언어 python™

Ch7. OOP (Object-Oriented Programming)

School of Information Convergence

Prof. Dong-Hyuk Im

광운대학교 KwangWoon University

Preview

- Object-Oriented Concept

- OOP in Python
  - Constructor
  - Inheritance
  - Polymorphism
  - Module

들어가기 앞서서..

- 객체 지향 프로그래밍
  - 객체(object)는 함수와 변수를 하나의 단위로 묶 을 수 있는 방법이다. 이러한 프로그래밍 방식을 객체지향(object-oriented)이라고 한다.

Object-Oriented Framework

- Two basic programming paradigms:
  - Procedural
    - Organizing programs around functions or blocks of statements which manipulate data.
  - Object-Oriented
    - combining data and functionality and wrap it inside what is called an object.

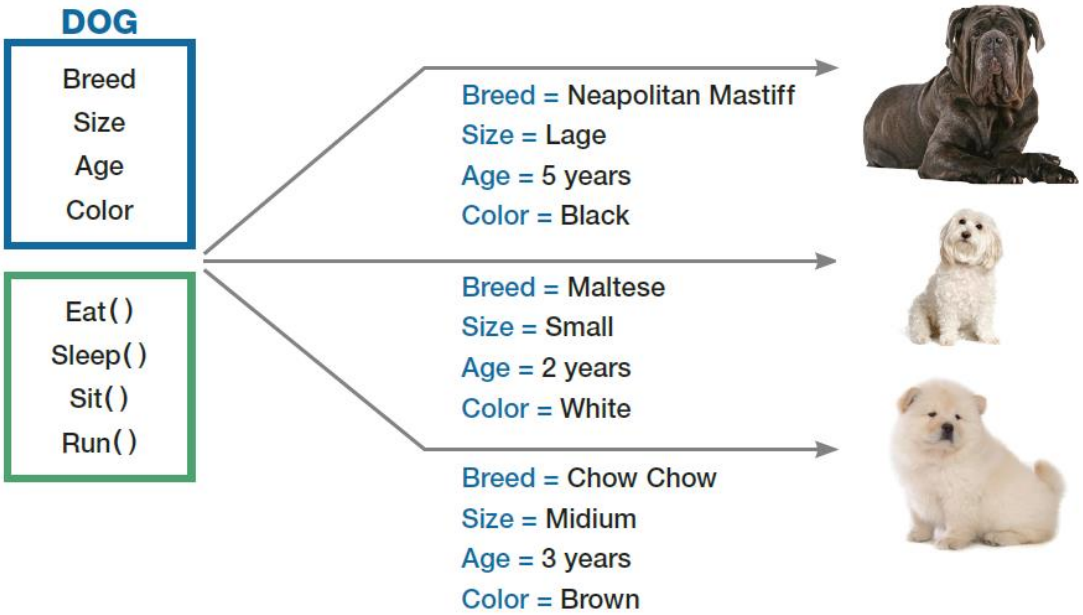
OOP Principles

- ***encapsulation***: hiding design details to make the program clearer and more easily modified later
- ***modularity***: the ability to make objects stand alone so they can be reused (our modules). Like the math module
- ***inheritance***: create a new object by inheriting (like father to son) many object characteristics while creating or over-riding for this object
- ***polymorphism***: (hard) Allow one message to be sent to any object and have it respond appropriately based on the type of object it is.

Class and Object

- Classes and objects are the two main aspects of object oriented programming.
- A **class** creates a new *type*.
- Where **objects** are *instances* of the class.
- Python doesn't use separate class interface definitions as in some languages
- You just define the class and then use it

*객체와 클래스



붕어빵틀
(클래스)

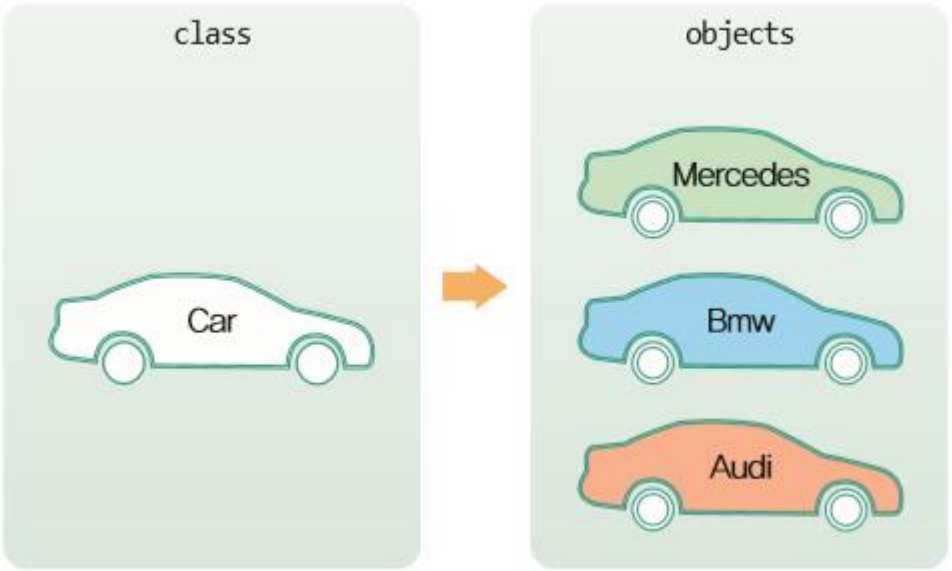
붕어빵
(인스턴스)

Filed and Methods

- Objects can store data using ordinary variables that *belong* to the object.
- Variables that belong to an object or class are called as **fields**.
- Objects can also have functionality by using functions that *belong* to the class. Such functions are called **methods**.

Creating Object

- A class is created using the class keyword.
- The fields and methods of the class are listed in an indented block.



Creating Object (1/2)

```
class Car:
    def drive(self):
        self.speed = 10

myCar = Car()
myCar.color = "blue"
myCar.model = "E-Class"

myCar.drive()                    # 객체 안의 drive() 메소드가 호출된다.
print(myCar.speed)               # 10이 출력된다.
```

Creating Object (2/2)

```
class Car:
    def drive(self):
        self.speed = 60

myCar = Car()
myCar.speed = 0
myCar.model = "E-Class"
myCar.color = "blue"
myCar.year = "2017"

print("자동차 객체를 생성하였습니다.")
print("자동차의 속도는", myCar.speed)
print("자동차의 색상은", myCar.color)

print("자동차의 모델은", myCar.model)
print("자동차를 주행합니다.")
myCar.drive()
print("자동차의 속도는", myCar.speed)
```

```
자동차 객체를 생성하였습니다.
자동차의 속도는 0
자동차의 색상은 blue
자동차의 모델은 E-Class
자동차를 주행합니다.
자동차의 속도는 60
```

The self

- The first argument of every method is a reference to the current instance of the class
- By convention, we name this argument ***`self`***
- In `__init__`, *`self`* refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called
- Similar to the keyword *this* in Java or C++
- But Python uses *self* more often than Java uses *this*

The self

- Although you must specify *self* explicitly when *defining* the method, you don't include it when *calling* the method.
- Python passes it for you automatically

Defining a method:

(this code inside a class definition.)

```
def set_age(self, num):
  self.age = num
```

Calling a method:

```
>>> x.set_age(23)
```

* 파이썬에서의 _ (1개)와 __(2개)

- _ (1개)의 쓰임
  - 이후로 사용하지 않을 변수에 특별한 이름을 부여하고 싶지 않을 때 사용

```
for _ in range(10):
    print("Hello, World")
```

- __(2개)의 쓰임
  - 특수한 예약 함수나 변수에 사용
  - 대표적으로 __str__ 이나 __init__() 함수가 있음

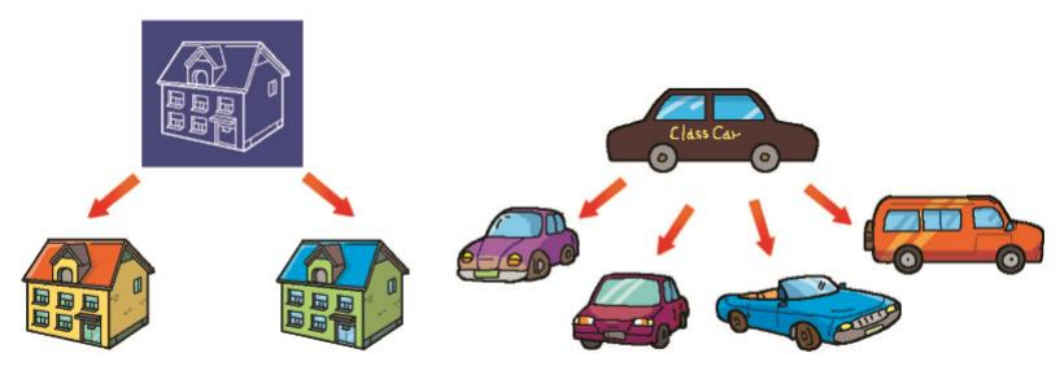
Constructor: __init__

- __init__ is called immediately after an instance of the class is created.

```
class Car:
    def __init__(self, speed, color, model):
        self.speed = speed
        self.color = color
        self.model = model

    def drive(self):
        self.speed = 60

myCar = Car(0, "blue", "E-class")
```

*하나의 클래스로 객체는 많이 만들 수 있다

```
class Car:
    def __init__(self, speed, color, model):
        self.speed = speed
        self.color = color
        self.model = model

    def drive(self):
        self.speed = 60

dadCar = Car(0, "silver", "A6")
momCar = Car(0, "white", "520d")
myCar = Car(0, "blue", "E-class")
```



Method __str__()

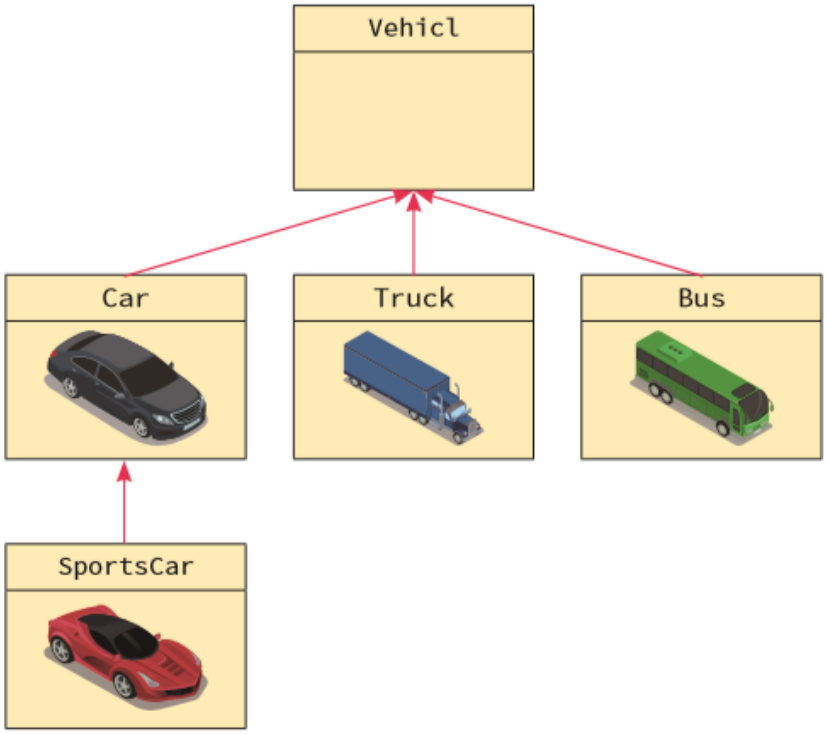
```
class Car:
    def __init__(self, speed, color, model):
        self.speed = speed
        self.color = color
        self.model = model

    def __str__(self):
        msg = "속도:"+ str(self.speed)+ " 색상:"+ self.color+ " 모델:"+ self.model
        return msg

myCar = Car(0, "blue", "E-class")
print(myCar)
```

속도:0 색상:blue 모델:E-class

Inheritance

- One of the major benefits of object oriented programming is **reuse** of code
- One of the ways this is achieved is through the inheritance mechanism.
- It refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.

Inheritance



Python Inheritance Syntax

class BaseClass:

  Body of base class

class DerivedClass(BaseClass):

  Body of derived class

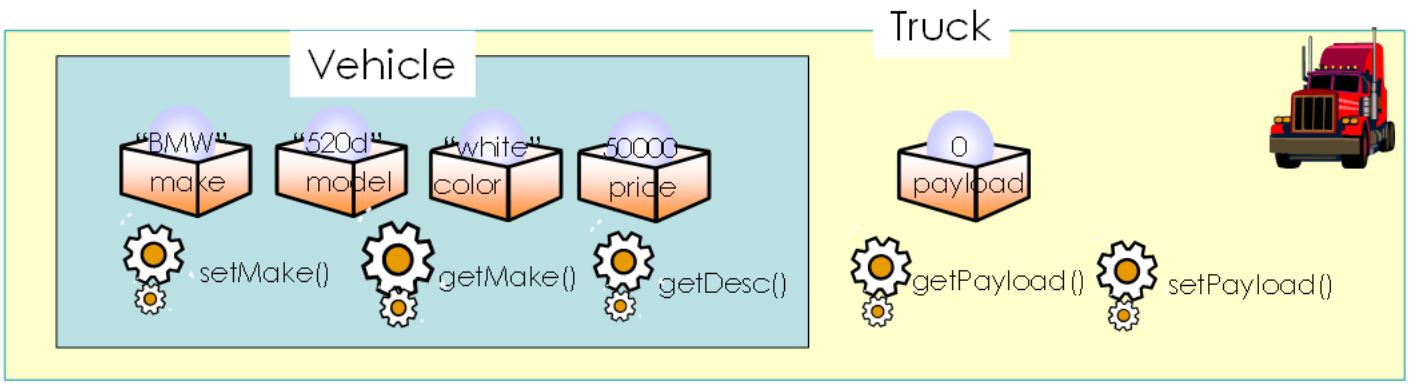
Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

*Lab: 상속 예제

```
# 일반적인 운송수단을 나타내는 클래스이다.
class Vehicle:
        def __init__(self, make, model, color,  price):
                self.make = make                # 메이커
                self.model = model    # 모델
                self.color = color              # 자동차의 색상
                self.price = price              # 자동차의 가격

        def setMake(self, make):        # 설정자 메소드
                self.make = make

        def getMake(self):              # 접근자 메소드
                return self.make

        # 차량에 대한 정보를 문자열로 요약하여서 반환한다.
        def getDesc(self):
                return "차량 =("+str(self.make)+","+\
                  str(self.model)+","+\
                  str(self.color)+","+\
                  str(self.price)+")"
```

*Lab: 상속 예제

```
class Truck(Vehicle) :                                  # ①
        def __init__(self, make, model, color, price, payload):
                super().__init__(make, model, color, price)      # ②
                self.payload=payload                             # ③

        def setPayload(self, payload):          # 설정자 메소드
                self.payload=payload

        def getPayload(self):                   # 접근자 메소드
                return self.payload
```

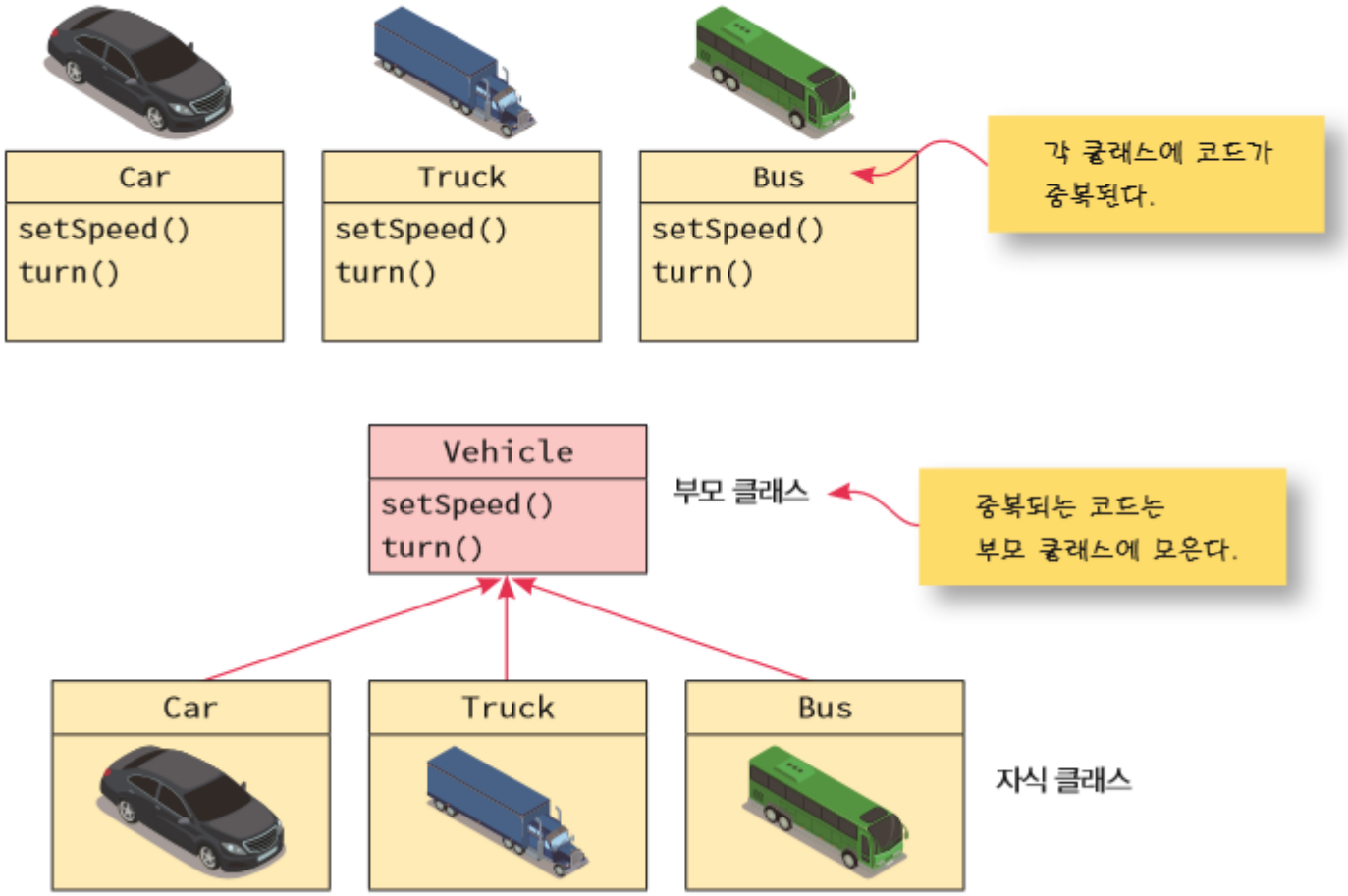


*Lab: 상속 예제

```
def main():                                          # main() 함수 정의
        myTruck = Truck("Tisla", "Model S", "white", 10000, 2000)
        myTruck.setMake("Tesla")                     # 설정자 메소드 호출
        myTruck.setPayload(2000)                     # 설정자 메소드 호출
        print(myTruck.getDesc())                     # 트럭 객체를 문자열로 출력


main()
```

차량 =(Tesla,Model S,white,10000)

*왜 상속을 사용하는가??



Extending __init__

- Same as for redefining any other method…
  - Commonly, the ancestor's **`__init__`** method is executed in addition to new commands.
  - You'll often see something like this in the **`__init__`** method of subclasses:

```
parentClass.__init__(self, x, y)
```

where parentClass is the name of the parent's class.

```
class ChildClass(ParentClass) :
        def __init__(self):
                super().__init__()
                ...
```

*Lab: 학생과 강사

- 일반적인 사람을 나타내는 Person 클래스를 정의한다. Person 클래스를 상속받아서 학생을 나타내는 클래스 Student와 선생님을 나타내는 클래스 Teacher를 정의한다.

```
이름=홍길동
주민번호=12345678
수강과목=['자료구조']
평점=0
이름=김철수
주민번호=123456790
강의과목=['Python']
월급=3000000
```

Solution

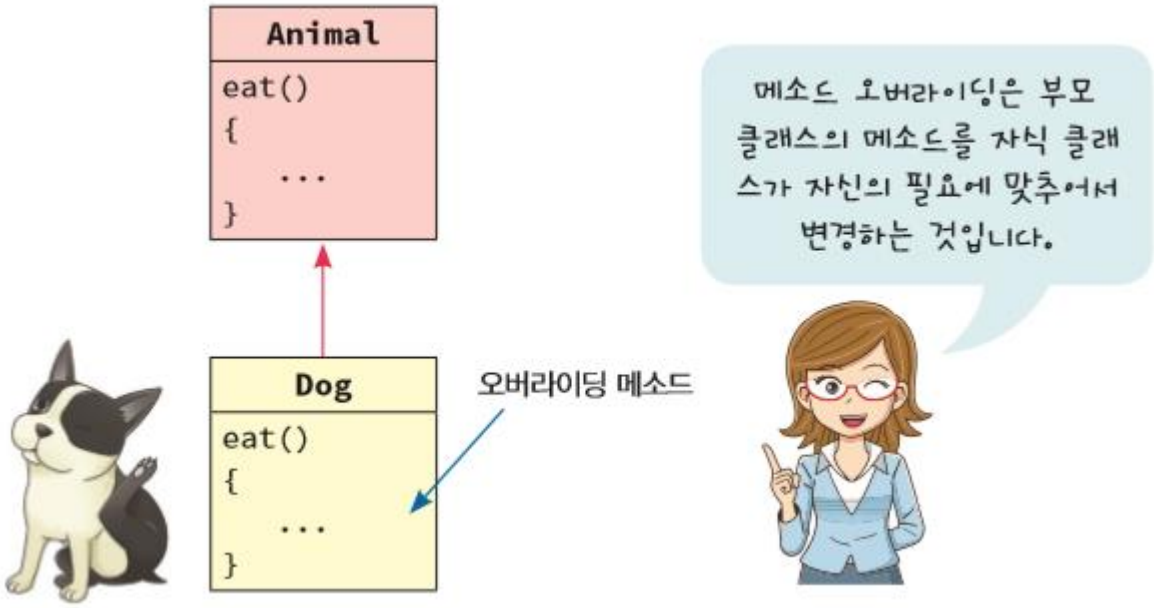
```
class Person:
    def __init__(self, name, number):
        self.name = name
        self.number = number

class Student(Person):
    UNDERGRADUATE=0
    POSTGRADUATE = 1

    def __init__(self, name, number, studentType ):
        super().__init__(name, number)
        self.studentType = studentType
        self.gpa=0
        self.classes = []

    def enrollCourse(self, course):
        self.classes.append(course)

    def __str__(self):
        return  "\n이름="+self.name+ "\n주민번호="+self.number+\
            "\n수강과목="+ str(self.classes)+ "\n평점="+str(self.gpa)
```

Solution

```
class Teacher(Person):
    def __init__(self, name, number):
        super().__init__(name, number)
        self.courses = []
        self.salary=3000000

    def assignTeaching(self, course):
        self.courses.append(course)

    def __str__(self):
        return  "\n이름="+self.name+ "\n주민번호="+self.number+\
            "\n강의과목="+str(self.courses)+ "\n월급="+str(self.salary)

hong = Student("홍길동", "12345678", Student.UNDERGRADUATE )
hong.enrollCourse("자료구조")
print(hong)

kim  = Teacher("김철수", "123456790")
kim.assignTeaching("Python")
print(kim)
```

Altering the Behavior of Inherited Methods: Overriding

- **Override:** To redefine how inherited method of base class works in derived class
- Two choices when overriding
  - Completely new functionality vs. overridden method
  - Incorporate functionality of overridden method, add more



Example

```
class Animal:
    def __init__(self, name=""):
        self.name=name
    def eat(self):
        print("동물이 먹고 있습니다. ")

class Dog(Animal):
    def __init__(self):
            super().__init__()
    def eat(self):
        print("강아지가 먹고 있습니다. ")

d = Dog();
d.eat()
```

강아지가 먹고 있습니다.

Understanding Polymorphism

- **Polymorphism:** Aspect of object-oriented programming that allows you to send same message to objects of different classes, related by inheritance, and achieve different but appropriate results for each object

*다형성의 의미

- **다형성(polymorphism)**은 **“많은(poly)+모양(morph)**“이라는 의미로서 주로 프로그래밍 언어에서 하나의 식별자로 다양한 타입(클래스)을 처리하는 것을 의미한다.

* 다형성의 사용 예

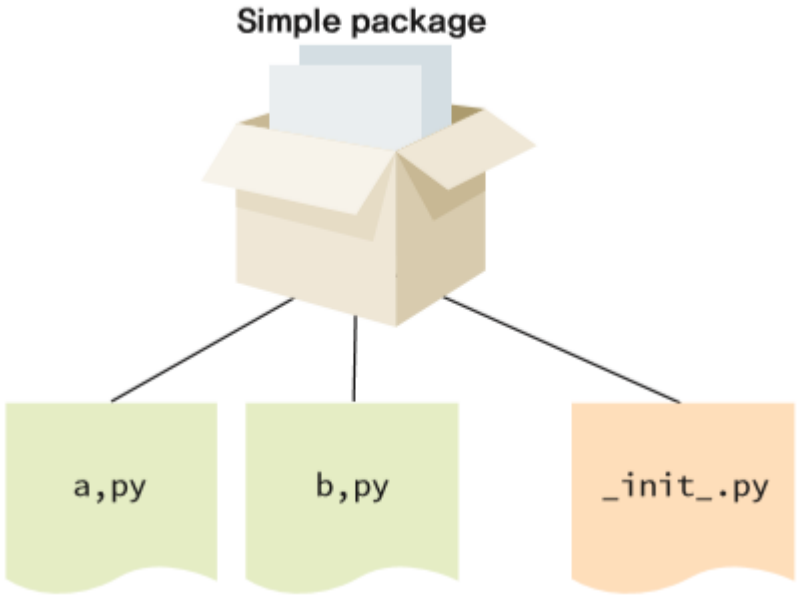
```
>>> list = [1, 2, 3]                # 리스트
>>> len(list)
3

>>> s = "This is a sentense" # 문자열
>>> len(s)
18

>>> d = {'aaa': 1, 'bbb': 2}     # 딕셔너리
>>> len(d)
2
```

Example

```
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        return '알 수 없음'

class Dog(Animal):
    def speak(self):
        return '멍멍!'

class Cat(Animal):
    def speak(self):
        return '야옹!'

animalList = [Dog('dog1'),
              Dog('dog2'),
              Cat('cat1')]

for a in animalList:
    print (a.name + ': ' + a.speak())
```

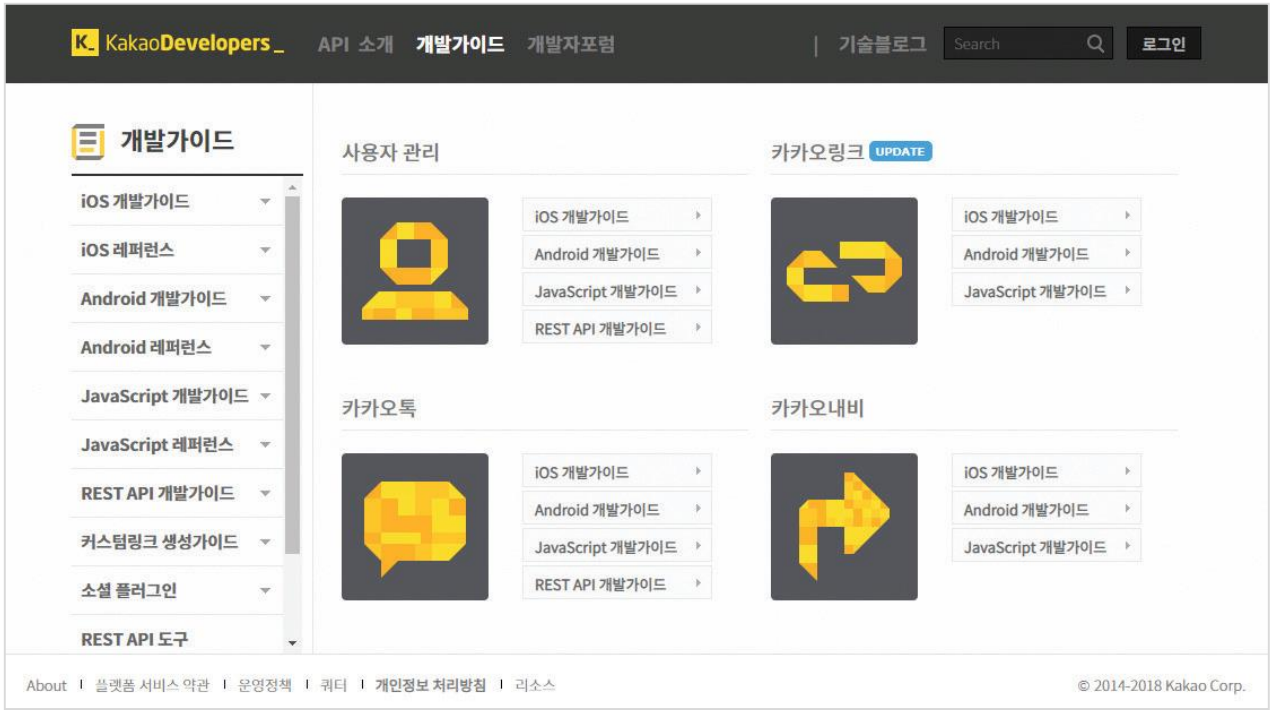
*모듈이란

- 함수나 변수들을 모아 놓은 파일을 모듈(module)



*모듈의 개념

- 모듈화된 프로그램을 사용하면, 다른 개발자가 만든 프로그램이나 자신이 만든 프로그램을 매우 쉽게 사용하거나 제공할 수 있다



[Kakao Developers의 개발가이드]

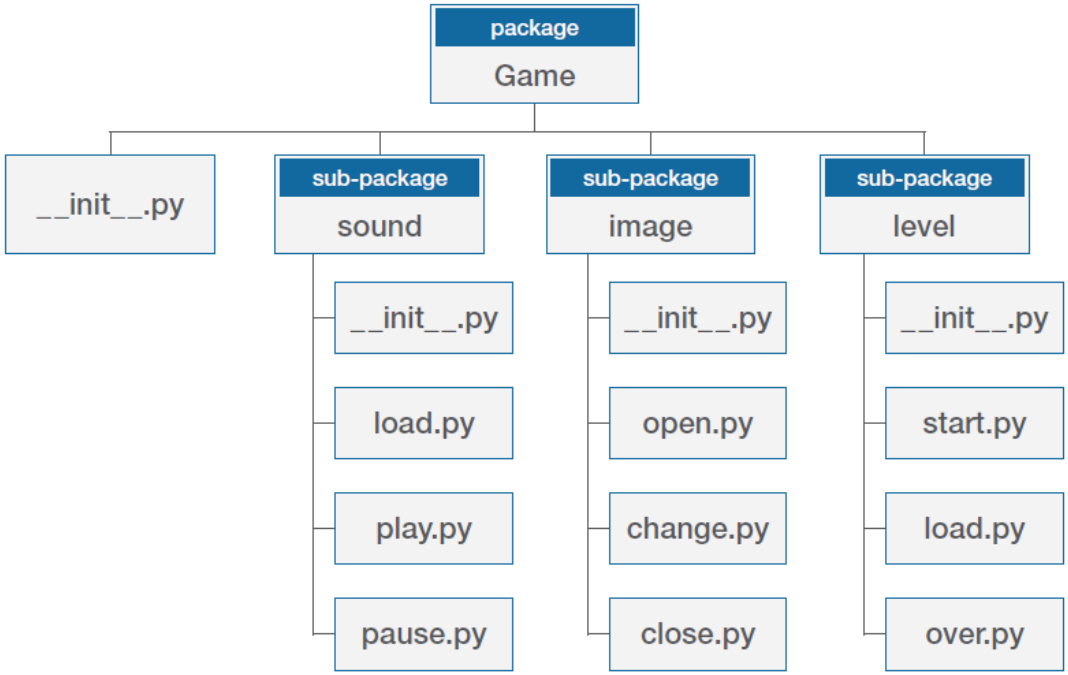
*모듈의 개념

- 내장 모듈이라고 하여 파이썬에서 기본적으로 제공하는 모듈 중 대표적으로 random 모듈이 있다. 이는 난수를 쉽게 생성해 주는 모듈이다. random 모듈을 호출하기 위한 코드는 다음과 같다.

```
>>> import random
>>> random.randint(1, 1000)
198
```

*패키지의 개념

- 패키지(packages)는 모듈의 묶음이다. 일종의 디렉토리처럼 하나의 패키지 안에 여러 개의 모듈이 있는데, 이 모듈들이 서로 포함 관계를 가지며 거대한 패키지를 만든다



[모듈과 패키지의 관계]

모듈 작성

fibo.py

```
# 피보나치 수열 모듈


def fib(n):    # 피보나치 수열을 화면에 출력한다.
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()
```

모듈 사용

```
>>> import fibo

>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

>>> fibo.__name__
'fibo'
```

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

If __name__ == “__main__”의 의미

```
#mod1.py

def sum(a, b):
    return a+b

def safe_sum(a, b):
    if type(a) != type(b):
        print(“더할 수 있는 것이 아닙니다“)
        return
    else:
        result = sum(a, b)
    return result
if __name__ == “__main__”:
    print(safe_sum(‘a’, 1))
    print(safe_sum(1, 4))
    print(safe_sum(10, 10.4))
```

```
C:\Python>python  mod1.py
더할 수 있는 것이 아닙니다
None
5
더할 수 있는 것이 아닙니다
None
```

*파일을 직접 실행하면 __name__ == “__main__”이 참이 되어 if문을 수행. 반대로 대화형 인터프리터나 다른 파일에서 이 모듈을 불러서 사용할 때는 거짓이 된다