Order

1. OOP & Class definition

2. Magic methods(built in methods)

3. __new__ vs __init__

4. method

5. Name space

6. Super

7. Inheritance => DRY, And *abstraction* is the key to understanding
=> inheritance
=> + diamond inheritance

8. Decorator

9. @property, setter, deleter

Why we use Class in Python ?

Firstly, What is OOP anyway?

Which I meet every time when I search Class

Object-oriented programming (**OOP**) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as *attributes;* and code, in the form of procedures, often known as *methods.* A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self"). In OOP, computer programs are designed by making them out of objects that interact with one another.[1][2] There is significant diversity of OOP languages, but the most popular ones are class-based, meaning that objects are instances of classes, which typically also determine their type.

" everything in Python is an object, even classes themselves"

" OOP란 프로그램을 단순히 데이터와 처리 방법으로 나누는 것이 아니라, 프로그램을 수많은 '객체'라는 기본 단위로 나누고 이 객체들의 상호작용으로 서술하는 방식이다.
객체를 데이터의 묶음으로만 착각하기 쉬운데, 그보다는
하나의 '역할'을 수행하는 메소드와 데이터의 묶음으로 봐야 한다. "

OOP(객체지향 프로그래밍), Class

=> 객체에 data, properties를 잘 보유하고
=> class 를 통해서 관계를 맺고 효율적으로 확장해 나가고

Then => What is Class ?

"Simply, a logical grouping of data and functions"

=> Logically grouping information is a way to group information in a logical manner. For instance, if you were listing cars, you wouldn't list the makes and the models in the same `<select>` list, as it isn't logical. You would have a separate `<select>` for each one. => 논리적인 메카니즘으로 정보를 그룹핑

So why we use Class?

- 우리가 하는 모든 것에 효율성이란 것을 추구해야 할 의무가 있다면 Python에서 대부분의 경우 우리는 Class를 사용해야 한다고 생각.

- Class : blueprint == 하나하나의 독립적 개체를 찍어내는 청사진 => 구조를 제공 => 그 자체가 인스턴스 자체를 만들어내건 아님.

- 자동차를 -> 기본 기능과 구조는 같지만 각각의 여러가지 특성들로 인해 종류가 수만가지인 개체로 분류 되어진다고 가정 해보자.

- 어떻게 만들 것인가? By Using Class ! (This is why we use class)
  1. 공유해야 할 Base 공유 => DRY
  2. 각각 개체(Instance)의 독립적 성격 지켜주며, 얼마든지 확장 가능.
  =>Base를 기반으로 수만가지의 개체를. 얼마든지 만들수 있다.

That's why we use Class in Python

*** [객체와 인스턴스의 차이] == [Object vs Instance]**

클래스에 의해서 만들어진 객체를 인스턴스라고도 한다. 그렇다면 객체와 인스턴스의 차이는 무엇일까?

이렇게 생각해 보자. a = Cookie() 이렇게 만들어진 a는 객체이다. 그리고 a라는 객체는 Cookie의 인스턴스이다. 즉, 인스턴스라는 말은 특정 객체(a)가 어떤 클래스(Cookie)의 객체인지를 관계 위주로 설명할 때 사용된다. 즉, "a는 인스턴스" 보다는 "a는 객체"라는 표현이 어울리며, "a는 Cookie의 객체" 보다는 "a는 Cookie의 인스턴스"라는 표현이 훨씬 잘 어울린다.

" Class를 보다보니 꼭 항상 나오는게 있다 "

__init__ ( ) => ? => 생성자 (Constructor)

`Python에서` 메소드 이름을 `__init__`으로 정해주면
=> 해당 메소드는 생성자로 인식되어 인스턴스가 생성되는 시점에 자동으로 호출된다.

- self(=instance)를 첫번째 인자로 받는다.
- 대개 변수에 인자를 전달하는 역할을 한다고 이해하면 된다.
- 근본적으로는 해당 클래스를 통해 instance 를 만들게 되면 초기에 실행 되는 함수라고 생각하면 된다. => initailizer

" 아하! "

" 근데 "

" 앞뒤로 언더바 두개는 뭐지? "

Dunder or magic **methods** in **Python** are the **methods** having two prefix and suffix underscores in the **method** name. **Dunder** here means “Double Under (Underscores)”. These are commonly used for operator overloading. Few examples for magic **methods** are: __init__, __add__, __len__, __repr__ etc.

=> Magic method ?

- https://rszalski.github.io/magicmethods/ <- 다양한 magic method들이 있다.

=> Meaning, Built-in functions “처럼” Python에서 기본 제공하는 함수들.

=> Can define (원래의 성격 그대로든, +a의 형태로 변형해서든)해서 쓸수있다.

=> magic method의 대부분은 Built-in function으로부터 나오고 있다.

It also means, **“built-in function이 magic method인 것은 아니다.”**

당연한 것이 method == function인 것은 아니니까

=>교집합을 이루고 있는 것들 즉, magic method 이면서 built-in function에서 나온 것들은

=> 우리가 정의하는 Class에 내부에 정의하므로 해당 Class의 Built-in type function 으로서 편하게 쓸수 있다.

“ Instance.__len__() 으로 쓰지 않고, len(Instance) 이렇게도 쓸수 있다는 말 “

* def __len__(self):

```
        return len(self.a)    <= 그대로 쓴다는건 이런식을 의미한다
```

That’s part of the power of magic methods. The vast majority of them allow us to define meaning for operators so that we can use them on our own classes just like they were built in types.

근데, 이경우를 한번 살펴보자

```
def __add__(self, other):
    return self.pay + other.pay
```

```
emp_1 = Employee('John', 'Smith', 50000)
emp_2 = Employee('Kim', 'hoon', 42500)
```

```
print(1, add(emp_1, emp_2))
```

```
    print(2, emp_1.add(emp_2))
AttributeError: 'Employee' object has no attribute 'add'
```

```
print(2, emp_1.add(emp_2))
```

```
print(1, add(emp_1, emp_2))
NameError: name 'add' is not defined
```

```
print(3, emp_1.__add__(emp_2))
```

```
3 92500
```

세번째만 제대로 돌아간다.
=>__new__, __init__과 같은 것은 class 생성과 관련해 특수한 기능을 하는 것으로 그 효용가치가 있다고 판단되고,
=> built-in funciton에서 온 것들은 호출시 built-in function처럼 편하게 쓸수 있다는 이점을 가지고 있다고 하자
=>근데 위의 결과라면__add__()와 같은 것은 위의 두개에도 해당하지 않고, 그것이 주는 이점이 무엇인지를 모르겠다. 그냥 정의해서 쓰는 것과 무슨 차이를 갖는 걸까
=> magic method를 class 안에서 class의 사용을 좀더 용이하게 만들어주는 것이라고 이해하자,
=> __add__ => ?
=> instance간 +(덧셈)는 성립이 되지 않는다. 그냥 주소값 두개를 어떻게 더하겠는가?
속성은 class안에 담겨있어 따로 속성명을 알아서 접근의 과정을 거쳐서 접근 해야하고
=> __add__() 를 통해 그것을 가능한 형태로 만들수 있다.
=> 인스턴스 자체를 인자로 받아 인스턴스들 내부의 어떤 속성간에 대한 덧셈 연산을 하는데 쓰인다.
=>편리 => 효용가치가 있다!이렇게 다른 효용가치로 편리함을 주는 다양한 magic method들이 있다.
그리고 대부분은 built-in으로 부터 온것이 많다.

```
class math:

    def __init__(self, a):
        self.a = a

    def __add__(self, other):
        return self.a + other.a

a = math(10)
b = math(25)

print(a + b)
```

```
/Users/imac/Desktop/
35
```

이렇게 다른 효용가치로 편리함을 주는 다양한 magic method들이 있다. 그리고 그것들의 대부분은 built-in function 으로 부터 온것이 많다.

“__init__을 조사하다 보니 계속 만나게 되는 __new__” => ?

__new__ vs __init__

- 일단 두함수는 모두 => magic method => You can see dunder

그리고 사실,

- Usually it’s uncommon to override __new__ method, but sometimes it is required if you are writing APIs or customizing class or instance creation or abstracting something using classes.

- It has to be overridden, if we don’t have any limitation for the class. The point is new is called implicitly **before** Called init

Differences between __new__ and __init__

#

__new__ => handles object creation

__init__ => handles object initialization

=> __new__ invoked previously

#

__new__ accepts *cls* as it's first parameter

__init__ accepts *self* as it’s first parameter

=> because when calling __new__ you actually don't have an instance yet, therefore no *self* exists at that moment, whereas __init__ is called after __new__ and the instance is in place, so you can use *self* with it.

Usage of Example)

1) => X (잘못된)

```
class A:

    def __new__(cls):
        print(“A.__new__ called”)

    def __init__(self):
        print (“A.__init__ called”) # -> is actually never called

=> A()
```

output is:

```
A.__new__ called => why is this only called?
```

=>

Obviously the instantiation is evaluated to None since we didn't return anything from the constructor.

2) => O

```
class Animal:
    def __new__(cls, *args, **kwargs):
        print('__new__() called.')
        print('args: ', args, ', kwargs: ', kwargs)
        return super().__new__(cls)

    def __init__(self, name):
        print('__init__() called.')
        self.name = name

a = Animal('Bob')
print(a.name)
```

output is:

```
__new__() called.
args:  ('Bob',) , kwargs:  {}
__init__() called.
Bob
```

__init__ is working !!

=>After the `__new__()` returning the created `Animal` object, Python will call `__init__()` automatically and pass the argument `Bob` to it.

" Return 으로 Object를 만들어냈고, 만들어 냈으니 initialize가 가능해진다

=> 근데 !

=> 아래와 같은 경우를 보게되었다 !

```
class AbstractClass(object):

    def __new__(cls, a, b):
        instance = super().__new__(cls)
        instance.__init__(a, b)
        return 3

    def __init__(self, a, b):
        print("Initializing Instance", a, b)

a = AbstractClass(2, 3)
print(a)
```

```
/Users/imac/PycharmProjects/untitled/venv/
Initializing Instance 2 3
3
```

=> __new__에서 instance (super().__new__(cls))를 return 안했는데? => ? =>

=> possib - 1) __new__에서 instance return => __init__정의

=> possib - 2) __new__에서 instance.__init__() 실행

두가지 모두 possible

Here you can see when we instantiate class it returns 3 instead of instance reference. Because we are returning 3 instead of created instance from __new__ method. We are calling __init__ explicitly. As I mentioned above, we have to call __init__ explicitly if we are not returning instance object from __new__ method.

The __new__ method is also used in conjunction with meta classes to customize class creation

메타클래스(metaclass)

=> 클래스를 만드는 클래스 => type() =>?=> Value의 타입이 뭔지 알려주는 함수아닌가 ?

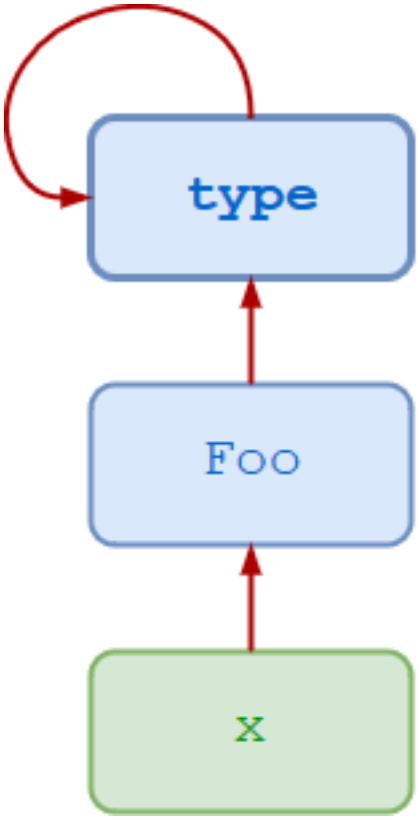
=> type(3) => <class 'int'> => 그러니까 ?

=> type(type) => <class 'type'>

=> class type => metaclass의 기능을 가지고 있다.

대략적으로 아래와 같은 느낌으로

- x is an instance of class Foo.
- Foo is an instance of the type metaclass.
- type is also an instance of the type metaclass, so it is an instance of itself.



논리) 쭉 올라가다 보니 클래스는 type에서 만들어졌고, 그래서 type은? 하고 찍어보니 type은 type자신으로 부터 만들어졌다. => type == class를 만들어내는 기준점의 class => class를 만들어내는 class => type == metaclass

사용방식

1. type을 사용하여 동적으로 클래스를 생성하는 방식

- 1) => type('class이름', (기반class,), {속성 메소드 = dict 형태로})

```
>>> temp = type('temp', (), {})
>>> temp
<class '__main__.temp'>
```

- **===> 변수 = type('class이름', (기반class,), 속성 메소드- **dict 형태로)**

```
>>> ins = type('temp', (object,), {'a':3, 'm':lambda a, b: a+b})
>>> print(ins.m(3,4))
7
```

2. type을 상속받아서 메타클래스 구현

then => def 생성Class (metaclass=myMetaclass(임의의 이름))

```
class myMetaclass(type):

    def __new__(cls, clsname, bases, dct):
        assert type(dct['a']) is int, 'a속성이 정수가 아니에요'
        return type.__new__(cls, clsname, bases, dct)

class Temp(metaclass=myMetaclass):
    a = 3.14

ins = Temp()
```

Output is => AssertionError: a속성이 정수가 아니에요

=> 지금 까지 들은 바로는 메타 클래스를 직접적으로 쓸 일은 많지 않다.
Django와 같은 프레임워크를 만들어야 한다면 필요한 정도라고 한다. 아무튼 느낌 Check

Method

=> method? => A **function** returns a value, but a procedure does not. A **method** is similar to a **function**, but is internal to part of a class.

bound => partially applied :

`foo` is just a function, but when you call `a.foo` you don't just get the function, you get a "partially applied" version of the function with the object instance `a` bound as the first argument to the function. `foo` expects 2 arguments, while `a.foo` only expects 1 argument.

Instance method

- No decorator
- 첫번째 인자로 self(just convention)를 갖는다. => 인스턴스 자신을 의미한다.
- 인스턴스를 통해 접근할 수 있다.
- Bounded to the instance
- 클래스를 통해서도 함수 자체에는 접근은 가능하다. => 주소값에 접근은 할 수 있다는 말

  => but

  => instance를 통한(self가 지칭해 주므로 인자없이 하는) direct 호출과는 다르게

  => 인자를 넣지 않고 그냥 실행 시킬 순 없다 (인자를 넣어야 한다고 에러가 뜬다)

  => 실행시키려면 인자에 해당 instance를 넣어줘야 한다.(instance를 지칭하는 것으로 있는 self의 자리에 엄밀히 말하면 self는 아니지만 instance를 넣어줌으로 충족시키는 것)

  => 그렇게 해주면

  => 해당 인스턴스에 대한 동작으로서 실행이 되어진다

  => 결국 이것은 instance를 통한 접근으로 보여진다.

[메서드의 또 다른 호출 방법]

잘 사용하지는 않지만 다음과 같이 메서드를 호출하는 것도 가능하다.

```
>>> a = FourCal()
>>> FourCal.setdata(a, 4, 2)
```

위와 같이 "클래스명.메서드" 형태로 호출할 때는 객체 a를 입력 인수로 꼭 넣어 주어야 한다. 반면에 다음처럼 "객체.메서드" 형태로 호출할 때는 첫 번째 입력 인수(self)를 반드시 생략해야 한다.

```
>>> a = FourCal()
>>> a.setdata(4, 2)
```

Class method

- Decorator - @classmethod
- 첫번째 인자로 cls(also just convention)를 갖는다. => class 자신을 의미한다.
- class를 통해 접근할 수 있다.
- Bounded to the class
- class에 대한 creation of instance를 필요로 하지 않는다.
- 인스턴스를 통해서 dot syntax로 주소에 접근을 한다고 해도 해당 class에 bound(소속) 되어 있는 method라는 설명과 주소 자체가 찍힐 뿐 해당 인스턴스 자체와는 직접적 관계(bouned)를 갖지 않는다.
- 하지만 실행은 된다. Name space에 의해 instance가 자신의 현재 문맥에서 실행 되어지는 능력을 갖지 못할 경우 자신의 상위 범위를 참조 하기 때문이다. Self -> Class -> Super
  => 그렇게 되면 자연히 instance.classmethod(x) => class.classmethod(o) 로 가기 때문에 실행이 되는 것.

Static method

- Decorator - @staticmethod
- 정해진 인자는 없다 (지정해 주면 받는다) - There is no difference with normal function
- class 안에 있긴 하지만
- Bounded to nothing(Neither class or instance)
- **class에 대한 creation of instance를 필요로 하지 않는다.**
- 클래스를 통해서든 인스턴스를 통해서든 접근이 가능하다.
- 실행 또한 어느 쪽으로 접근해서든 아무 문제없이 된다.
- 가독성과 효율성(해당 class와 관련하여 사용되는 것의 묶음으로 정리 된다는)면에서 Class안에 소속되는 것일 뿐, 일반 function과 다를게 없다고 생각하면 된다.

The difference between a static method and a class method is:

- Class method works with the class since its parameter is always the class itself.
- Static method knows nothing about the class and just deals with the parameters

```python
class Test:
    def ins(self):
        return 'instance method'

    @staticmethod
    def static():
        return 'static method'

    @classmethod
    def classM(cls):
        return 'class method'


a = Test()

print('-----------------------------------------------------------------------------------------------------------------------------------------------------------')
print('인스턴스를 통한 instance method 호출 : ', a.ins)
print('인스턴스를 통한 instance method 실행 : ', a.ins())
print('-----------------------------------------------------------------------------------------------------------------------------------------------------------')
print('인스턴스를 통한 staticmethod 접근 : ', a.static)
print('인스턴스를 통한 staticmethod 실행 : ', a.static())
print('-----------------------------------------------------------------------------------------------------------------------------------------------------------')
print('인스턴스를 통한 classmethod 접근 : ', a.classM)
print('인스턴스를 통한 classmethod 실행 : ', a.classM())
print('-----------------------------------------------------------------------------------------------------------------------------------------------------------')
print('클래스를 통한 instance method 접근 : ', Test.ins)
print('클래스를 통한 instance method 실행 by Test.ins()  : "TypeError: ins() missing 1 required positional argument"')
print('클래스를 통한 instance method 실행 by Test.ins(a) : ', Test.ins(a))
print('-----------------------------------------------------------------------------------------------------------------------------------------------------------')
print('클래스를 통한 staticmethod 접근 : ', Test.static)
print('클래스를 통한 staticmethod 실행 : ', Test.static())
print('-----------------------------------------------------------------------------------------------------------------------------------------------------------')
print('클래스를 통한 classmethod 접근 : ', Test.classM)
print('클래스를 통한 classmethod 실행 : ', Test.classM())
print('-----------------------------------------------------------------------------------------------------------------------------------------------------------')
```

```
/Users/imac/PycharmProjects/untitled/venv/bin/python /Users/imac/Desktop/class-project/making_class/class.py
-----------------------------------------------------------------------------------------------------------------------------------------------------------
인스턴스를 통한 instance method 호출 :  <bound method Test.ins of <__main__.Test object at 0x10b675080>>
인스턴스를 통한 instance method 실행 :  instance method
-----------------------------------------------------------------------------------------------------------------------------------------------------------
인스턴스를 통한 staticmethod 접근 :  <function Test.static at 0x10b674378>
인스턴스를 통한 staticmethod 실행 :  static method
-----------------------------------------------------------------------------------------------------------------------------------------------------------
인스턴스를 통한 classmethod 접근 :  <bound method Test.classM of <class '__main__.Test'>>
인스턴스를 통한 classmethod 실행 :  class method
-----------------------------------------------------------------------------------------------------------------------------------------------------------
클래스를 통한 instance method 접근 :  <function Test.ins at 0x10b6742f0>
클래스를 통한 instance method 실행 by Test.ins()  : "TypeError: ins() missing 1 required positional argument"
클래스를 통한 instance method 실행 by Test.ins(a) :  instance method
-----------------------------------------------------------------------------------------------------------------------------------------------------------
클래스를 통한 staticmethod 접근 :  <function Test.static at 0x10b674378>
클래스를 통한 staticmethod 실행 :  static method
-----------------------------------------------------------------------------------------------------------------------------------------------------------
클래스를 통한 classmethod 접근 :  <bound method Test.classM of <class '__main__.Test'>>
클래스를 통한 classmethod 실행 :  class method
-----------------------------------------------------------------------------------------------------------------------------------------------------------

Process finished with exit code 0
```

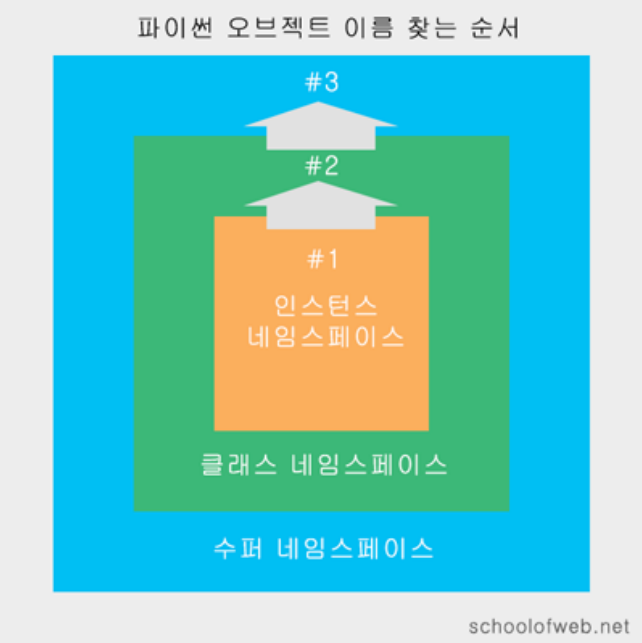
위에서

=> 네임 스페이스 ? Super ?

=> 한번 알아보자

1) 네임 스페이스 => 자식-> 부모

네임 스페이스 => 인스턴스 스페이스 -> 클래스 스페이스 -> 슈퍼 클래스 스페이스
만약에 탐색을 했는데 없다면 위의 순서대로 상위계층을 탐색해서 있으면 내놓는다.
(위에서 언급)



조금 더 들어가보자 => Name ?

What is Name in Python?

Name (also called identifier) is simply a name given to objects.
Everything in Python is an object. Name is a way to access the underlying object.
(<- You again hah?)

*** !!! name , memory, RAM !!! ***(짚고 넘어가기 좋은 기본 개념)

=> For example, when we do the assignment `a = 2`, here `2` is an object stored in memory and `a` is the name we associate it with. We can get the address (in RAM) of some object through the built-in function, `id()`. Let's check it.

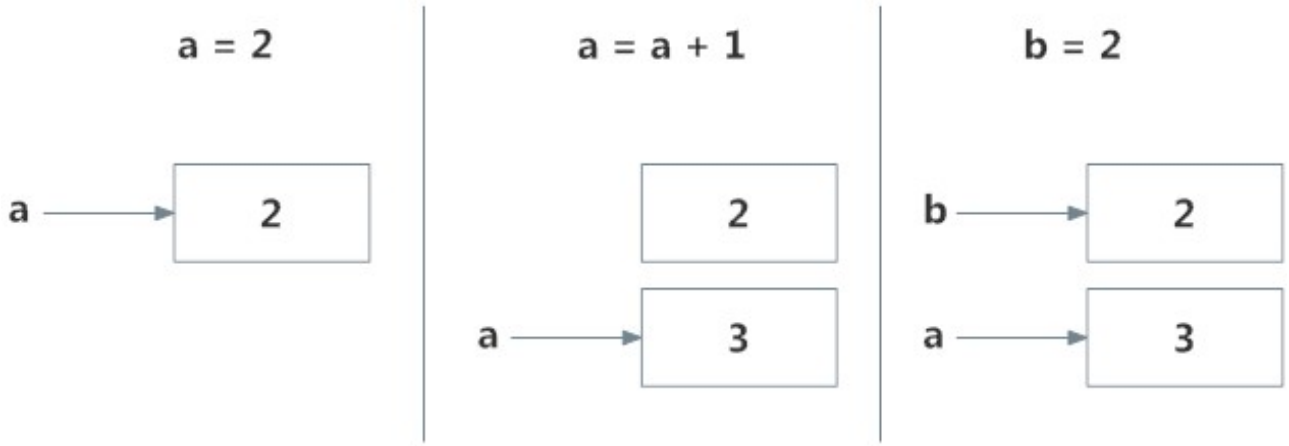
```
1   # Note: You may get different value of id
2
3   a = 2
4
5   print('id(a) =', id(a))
6   Output: id(a) = 10919424
7
8   a = a+1
9
10  print('id(a) =', id(a))
11  # Output: id(a) = 10919456
12
13  print('id(3) =', id(3))
14  # Output: id(3) = 10919456
15
16  b = 2
17
18  print('id(2) =', id(2))
19  # Output: id(2)= 10919424
```

```
print(id(2))
a = 2
b = 2
print(id(a))
print(id(b))
```

```
4338805872
4338805872
4338805872
```

*같은 value에 대해서
주소값은 같다

What is happening in the above sequence of steps? A diagram will help us explain this.



Initially, an object 2 is created and the name a is associated with it, when we do `a = a+1`, a new object 3 is created and now `a` associates with this object.

Note that `id(a)` and `id(3)` have same values.

Furthermore, when we do `b = 2`, the new name `b` gets associated with the previous object 2.

This is efficient as Python doesn't have to create a new duplicate object. This dynamic nature of name binding makes Python powerful; a name could refer to any type of object.

So, What is a Namespace in Python?

So now that we understand what names are, we can move on to the concept of namespaces.

To simply put it, namespace is a collection of names.

In Python, you can imagine a namespace as a mapping of every name, you have defined, to corresponding objects.

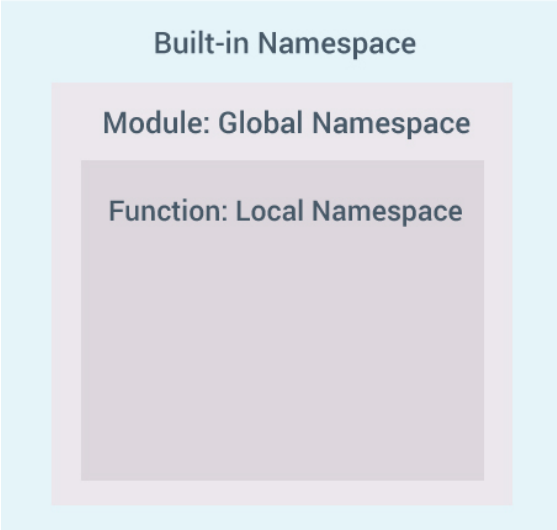
Different namespaces can co-exist at a given time but are completely isolated.

A namespace containing all the **built-in names** **is created when we start the Python interpreter and exists as long we don't exit.**

This is the reason that built-in functions like `id()`, `print()` etc. are always available to us from any part of the program. Each module creates its own global namespace.

These different namespaces are isolated. Hence, the same name that may exist in different modules do not collide.

Modules can have various functions and classes. A local namespace is created when a function is called, which has all the names defined in it. Similar, is the case with class. Following diagram may help to clarify this concept.



2) super( )

=> simply, parent 쉽게 호출

=> Inheritance 에서 용이 => DRY => simply override

"[Super is used to] *return a proxy object that delegates method calls to a parent or sibling class of type. This is useful for accessing inherited methods that have been overridden in a class. The search order is same as that used by getattr() except that the type itself is skipped.*"

How Is the Super Function Used?

The super function is somewhat versatile, and can be used in a couple of ways.

Use Case 1: Super can be called upon in a single inheritance, in order to refer to the parent class or multiple classes without explicitly naming them. It's somewhat of a shortcut, but more importantly, it helps keep your code maintainable for the foreseeable future.

Use Case 2: Super can be called upon in a dynamic execution environment for multiple or collaborative inheritance. This use is considered exclusive to Python, because it's not possible with languages that only support single inheritance or are statically compiled.

In Python 3, the syntax for super is:

```
super().methoName(args)
```

Whereas the normal way to call super (in older builds of Python) is:

```
super(subClass, instance).method(args)
```

As you can see, the newer version of Python makes the syntax a little simpler. But, older version의 것에서는 첫번째 인자인 subClass 자리에 넣는 것의 바로 위 super class(그게 어디에 있든 얼마나 떨어져있든)를 호출하는 기능을 가지고 있다. 즉, 자기 자신을 인자로 넣어 자신의 바로 위 super를 호출하는 것 외에 subClass 자리에 어떤 인자를 넣어주냐에 따라서 넣어준 subClass의 바로위 super를 호출하는 것이 가능하다. 그래서 그러한 기능을 필요로 할때 유용하게 쓰인다. 물론 어떤 클래스가 단순히 자신의 바로위 super를 호출하는 용도에서는 새로운 버전이 훨씬 간결하기 때문에 더 많이 쓰인다.

In order to use the function properly, the following conditions must be met:

- The method being called upon by *super()* must exist
- Both the caller and callee functions need to have a matching argument signature
- Every occurrence of the method must include *super()* after you use it

Ex)

```
class MyParentClass(object):
    def __init__(self):
        pass
Without super()
class SubClass(MyParentClass):
    def __init__(self):
        MyParentClass.__init__(self)
With older super()
class SubClass(MyParentClass):
    def __init__(self):
        super(SubClass, self).__init__()
```

```
class MyParentClass():
    def __init__(self, x, y):
        pass
With new super()
class SubClass(MyParentClass):
    def __init__(self, x, y):
        super().__init__(x, y).
```

Again, this process is much more straightforward than the traditional method. In this case, we had to call the super function's __init__ method to pass our arguments.

super function with multi-level inheritance

As we have stated previously that Python super() function allows us to refer the superclass implicitly. But in the case of multi-level inheritances which class will it refer? Well, Python super() will always refer the immediate superclass. Also Python super() function not only can refer the `__init__()` function but also can call all other function of the superclass. So, in the following example, we will see that.

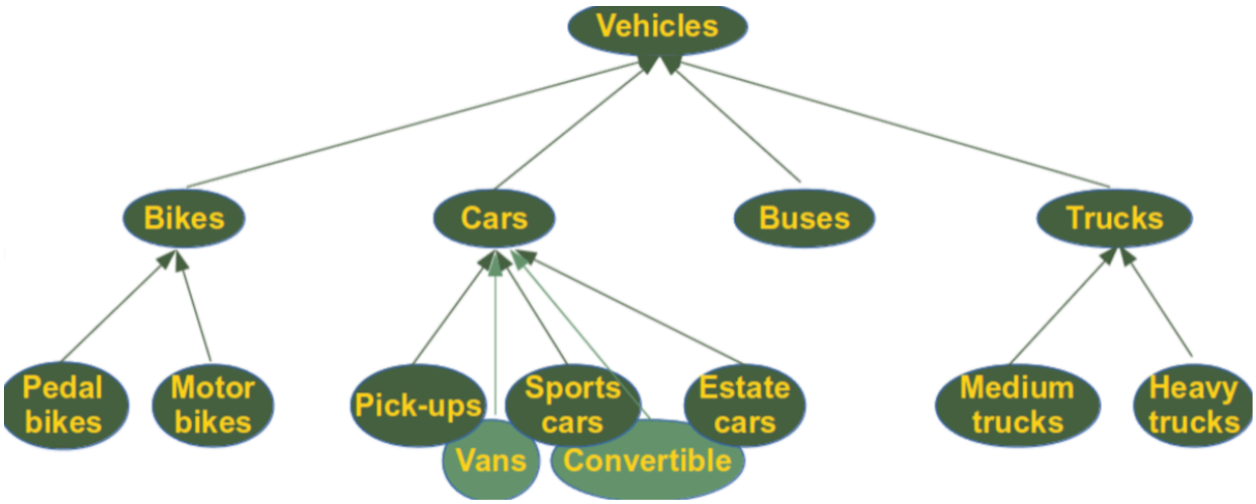
ex)

```
class A:
    def __init__(self):
        print('Initializing: class A')

    def sub_method(self, b):
        print('Printing from class A:', b)


class B(A):
    def __init__(self):
        print('Initializing: class B')
        super().__init__()

    def sub_method(self, b):
        print('Printing from class B:', b)
        super().sub_method(b + 1)


class C(B):
    def __init__(self):
        print('Initializing: class C')
        super().__init__()

    def sub_method(self, b):
        print('Printing from class C:', b)
        super().sub_method(b + 1)


if __name__ == '__main__':
    c = C()
    c.sub_method(1)
```

Output is :

```
/Users/imac/PycharmProjects/untitled/
Initializing: class C
Initializing: class B
Initializing: class A
Printing from class C: 1
Printing from class B: 2
Printing from class A: 3
```

Inheritance



Syntax)

```
class DerivedClassName(BaseClassName):
    pass
```

Just one simple example for explanation of Inheritance

```
class Person:

    def __init__(self, first, last, age):
        self.firstname = first
        self.lastname = last
        self.age = age

    def __str__(self):
        return self.firstname + " " + self.lastname + ", " + str(self.age)

class Employee(Person):

    def __init__(self, first, last, age, staffnum):
        super().__init__(first, last, age)
        self.staffnumber = staffnum

    def __str__(self):
        return super().__str__() + ", " +  self.staffnumber


x = Person("Marge", "Simpson", 36)
y = Employee("Homer", "Simpson", 28, "1007")

print(x)
print(y)
```

Output =>

```
"Marge Simpson 36"
"Homer Simpson 28 1007"
```

We have overridden the method __str__ from Person in Employee. By the way, we have overridden __init__ also. Method overriding is an object-oriented programming feature that allows a subclass to provide a different implementation of a method that is already defined by its superclass or by one of its superclasses. The implementation in the subclass overrides the implementation of the superclass by providing a method with the same name, same parameters or signature, and same return type as the method of the parent class.

Diamond Inheritance

=> 다이아몬드 상속 또한 super의 특성과 밀접하게 연관되어 있어 중복되는 느낌이 있지만

=> 중복을 handling 하는 부분과 관련하여 짚고 넘어가면 좋을 것 같다

Syntax)

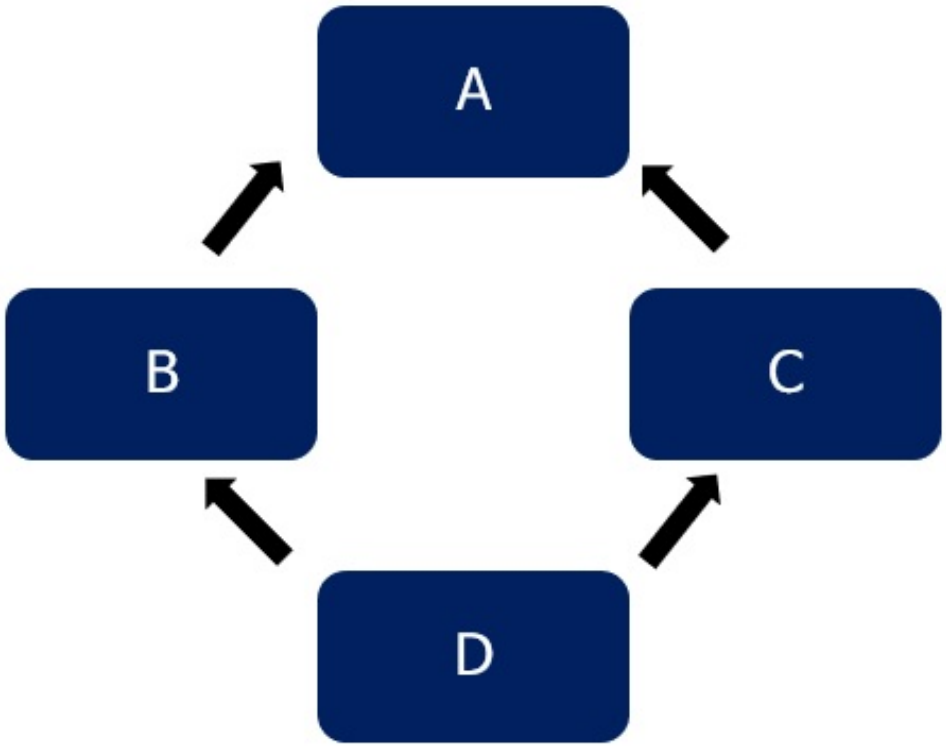
class Unknown(Human, Bird):

```
        def __init__(self):
            super().__init__()
            print("I'm Unknown. Please give me a name”)
```

=> Check Point

=> 중복되는 것이 있는 경우 상속인자의 class 순서에서 앞(왼쪽)의 것이 살아남는다.
예를들면 Human의 leg라는 class 변수가 4이고, Bird의 leg라는 class 변수가 2이면
Unknown은 다이아몬드 상속시 leg라는 변수에 대해 앞의 것인 4를 상속받게 된다는 의미
=> Python의 읽는 방향이 오른쪽에서 왼쪽이기 때문에
=> 오른쪽것을 먼저 읽고, 왼쪽것을 읽기 때문Override하기 때문

- 다이아몬드 상속이란 상속관계가 아래의 이미지와 같은 경우를 의미한다.

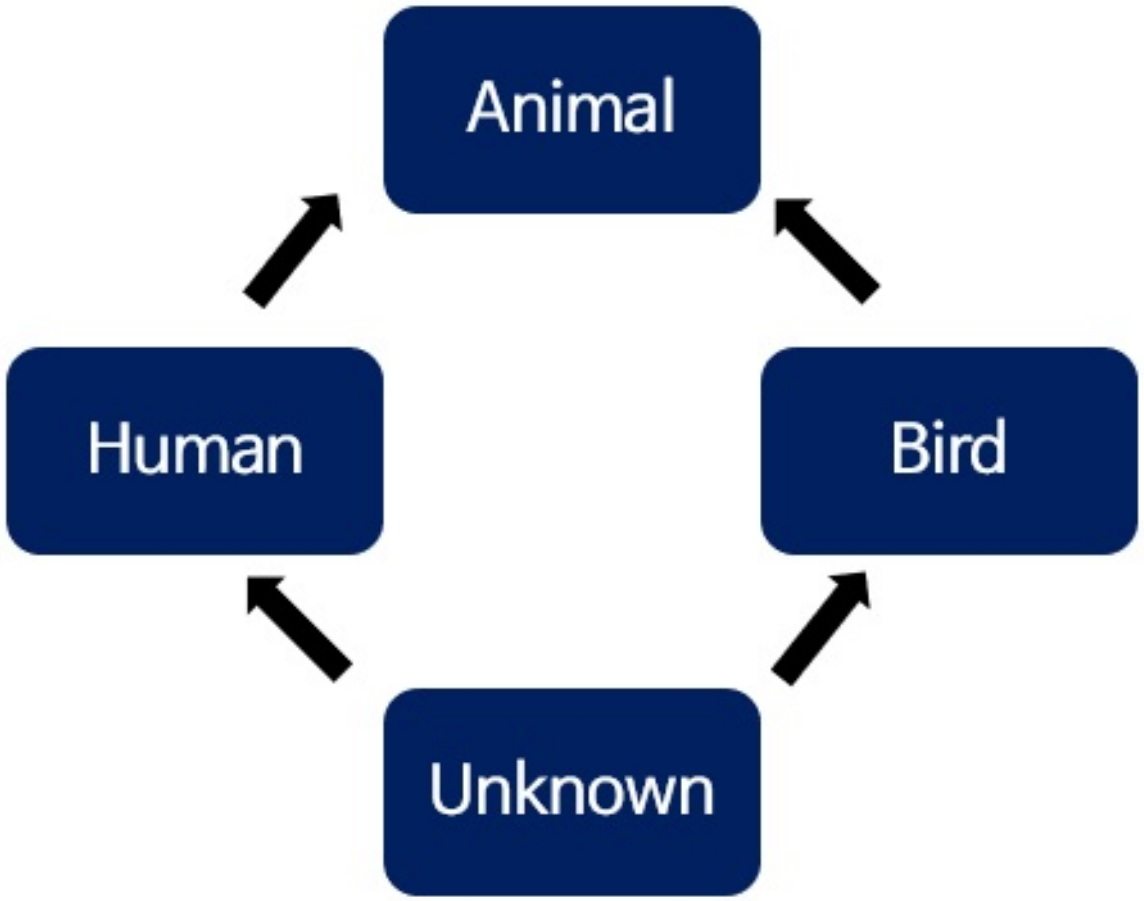


- 최상단의 클래스 A가 존재한다.
- 클래스 B와 클래스 C는 클래스 A를 상속받는다.
- 클래스 D는 클래스 B와 C (클래스 A를 상속받은) 를 상속받는다.

여기서 **문제의 여지가 있는 것은, 클래스 D이다.**

=> 클래스 D는 클래스 B와 C를 상속
=> 클래스 B와 C는 모두 클래스 A를 상속
=> 클래스 D가 클래스 B와 C를 상속받으면, 클래스 A를 2번 상속받는 셈.

" 흠… 한번 구체적인 예로 한번 들어가보자 "



=> Animal (동물)

Animal 상속 => Human, Bird

Human, Bird 상속 => Unknown

```
class Animal:
    def __init__(self):
        print("I'm an animal")
```

```
class Human(Animal):
    def __init__(self):
        Animal.__init__(self)
        print("I'm a human")
```

```
class Bird(Animal):
    def __init__(self):
        Animal.__init__(self)
        print("I'm a bird")

class Unknown(Human, Bird):
    def __init__(self):
        Human.__init__(self)
        Bird.__init__(self)
        print("I'm Unknown. Please give me a name")
```

```
instance = Unknown()
>>> 'I'm an animal'
>>> 'I'm a human'
>>> 'I'm an animal'
>>> 'I'm a bird'
>>> 'I'm Unknown. Please give me a name'
```

클래스 Animal 중복 상속 => I'm an animal 이 두번 print

=> Unknown은 Human 과 Bird 두가지 클래스를 상속

=> Human 도 Animal 상속, Bird 도 Animal 상속

=> Human 과 Bird 를 둘 다 상속받은 Unknown 은 Animal 을 두번 상속받은 꼴.

=> 이런 경우 위에서 정리한 super()를 이용

```
class Animal:
    def __init__(self):
        print("I'm an animal")

class Human(Animal):
    def __init__(self):
        super().__init__()
        print("I'm a human")

class Bird(Animal):
    def __init__(self):
        super().__init__()
        print("I'm a bird")
```

```
class Unknown(Human, Bird):
    def __init__(self):
        super().__init__()
        print("I'm Unknown. Please give me a name")
instance = Unknown()
```

```
>>> 'I'm an animal'
>>> 'I'm a bird'
>>> 'I'm a human'
>>> 'I'm Unknown. Please give me a name'
```

추가적으로, 상속 순서의 우선순위가

=> Human → Bird 에서

=> Bird → Human 으로 바뀌었음을 알 수 있다.

=> python의 읽는 순서가 right to left

=> overrided

=> 이러한 결과를 통해, 클래스명을 통해 부모클래스를 호출하는 것과 `super()` 메서드를 활용했을 때의 차이점을 알 수 있다.

코드의 순서그대로 출력을 원한다면 클래스명을 통해 메서드를 호출하는 것이 타당할 것이고,관계가 복잡해져 **중복호출을 예방**하기 위해서는 `super()` 메서드를 사용해야함을 알 수 있다.

인터페이스
그리고 다이아몬드 상속시 조금 tricky한 부분이 있었는데
=> 상속 class들의 인터페이스가 다른 경우
=> 피상속 class가 어떤 것을 따라야 할지 모르게 된다.
=> 그래서 error가 발생한다.

아래와 같은 경우는 피상속 class에서 상속 class의 인터페이스에 대한 인자 전달을 정확히 정해줘서 D가 제대로 동작 하는 경우이다. **kwargs의 속성도 살펴볼겸 이것의 논리를 한번 들여다보자.

```
class A(object):
    def __init__(self, a):
        self.a = a

class B(A):
    def __init__(self, b, **kw):
        self.b = b
        super(B, self).__init__(**kw)

class C(A):
    def __init__(self, c, **kw):
        self.c = c
        super(C, self).__init__(**kw)

class D(B,C):
    def __init__(self, a, b, c, d):
        super(D, self).__init__(a=a, b=b, c=c)
        self.d = d


d= D('ay', 'bee', 'cla', 'dra')
print(d.d)
```

=> dra

논리

=>D에 인자 4개를 넣어 인스턴스 d 생성
=>D의 상속 class B와 C의 __init__()에 들어온 인자들을 앞에서 순서대로 key값 a, b, c에 할당해서 전달
=> 1) C부터 읽음
=> C의 속성 c에 받은 인자 할당
=> C는 **kwargs의 속성에 따라 super A에게 D에게 받은 { a: ay, b: bee } 를 전달
=> A는 받은 a:ay의 ay를 자신의 a에 할당
=> 2) B를 읽음
=> B의 속성 b에 받은 인자 할당
=> B는 **kwargs의 속성에 따라 super A에게 D에게 받은 { a: ay, b: bee } 를 전달
=> A는 받은 a:ay의 ay를 자신의 a에 할당
=> 전달하지 않았던 그 마지막 인자를 자신의 속성 d에 할당.

- 인터페이스 구조가 같기 때문에 동작이 된다.
- 인자가 남으면 가변적으로 인자를 가지는 **kwargs 에게 넘겨주고, **kwargs는 key값에 맞게 인자를 받아서 사용하는 모습을 볼수 있다.(굉장히 자주 쓰이고 유용한 *args, **kwargs의 사용 예)

@decorator

decorator를 검색하니 => Higher-order function => ? => HOF

In mathematics and computer science, a **higher-order function** (also **functional**, **functional form** or **functor**)[*citation needed*] is a function that does at least one of the following:

- takes one or more functions as arguments (i.e. procedural parameters),
- returns a function as its result.

While not a purely functional language, Python supports many of the functional programming concepts, including functions as first-class objects.

First-Class Objects

In Python, functions are first-class objects. This means that **functions can be passed around and used as arguments**, just like any other object (string, int, float, list, and so on). Consider the following three functions:

=> ok => so what is decorator?

Put simply : **decorator wraps a function, modifying its behavior.**

decorator function without using @

==========

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

def say_whee():
    print("Whee!")

say_whee = my_decorator(say_whee)
```

```
The actual usage of @
```

==========

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_whee():
    print("Whee!")
```

What is the @decorator ?

=> 어떠한 function이 자신을 직접 수정하지 않으면서도
=> 그 function의 기능을 확장및 수정 할 수 있도록 도와주는 function의 적용을 시각적으로 표현한 것

=>So, decorator를 단다는 것의 의미

- decorator는 function을 인자로 가지는 Hof 인데 그 함수에
- @ + decorator를 자신의 머리위에 다는 'the 함수'를 decorator의 인자로 전달해서 확장한 version의 것으로서 쓴다는 의미

=>다중도 가능하다(function의 모듈화같은 느낌인듯)

```
@deco_a
@deco_b
@deco_c                          => deco_c -> deco_b -> deco_a 순서(아래 -> 위로)
def decorated( ):
        pass
```

=> class에도 @decorator를 다는 것이 가능하다.

`Simply, @classmethod, @staticmethod ?`

@classmethod => clasmethod(func) => making the func to a class method

@staticemthod => staticmethod(func) => making the func to a static method

Ex)

=>이렇게 우리가 봐오던 데로 decorator를 정의하고 @decorator를 달아서 쓰면 된다.

=> 여러개에도 얼마든지 쓸 수 있다.

```
def do_twice(func):
    def wrapper():
        func()
        func()
    return wrapper

@do_twice
def say_whee():
    print("Whee!")


@do_twice
def greet(name):
    print(f"Hello {name}")


say_whee()
greet("World!")
```

But =>

```
/Users/imac/PycharmProjects/untitled/venv/bin/python /Users/imac/Desktop/class-project/mak
Whee!
Traceback (most recent call last):
Whee!
  File "/Users/imac/Desktop/class-project/making_class/class.py", line 319, in <module>
    greet("World!")
TypeError: wrapper() takes 0 positional arguments but 1 was given

Process finished with exit code 1
```

=> error ?

=> ?

=> say_whee는 인자가 없고,

=> greet는 하나의 인자를 갖는다.

=> decorator를 통해 그것들을 handling 하는 wrapper function은 인자를 갖지 않고 그 내부에서 handling 하는 func에게도 어떤 인자도 전달하지 않는다.

=> 우리는 위의 decorator를 인자를 하나를 갖든, 갖지 않든 모두 handling 하고 싶다

=> *args and **kwargs allow you to pass a variable number of arguments to a function.

```python
def do_twice(func):
    def wrapper(*args):
        func(*args)
        func(*args)
    return wrapper

@do_twice
def say_whee():
    print("Whee!")


@do_twice
def greet(name):
    print(f"Hello {name}")


say_whee()
greet("World!")
```

```
/Users/imac/PycharmProjects/untitle
Whee!
Whee!
Hello World!
Hello World!

Process finished with exit code 0
```

* If you want to get “return value”, you should return *

예를 통해 말해보겠다.

```python
def do_twice(func):
    def wrapper(*args):
        func(*args)
        func(*args)
    return wrapper

@do_twice
def say_whee():
    print("Whee!")


@do_twice
def greet(name):
    print(f"Hello {name}")



say_whee()
greet("World!")


@do_twice
def return_greeting(name):
    print("Creating greeting")
    return f"Hi {name}"

return_greeting("Adam")
```

= 새로 정의한 return_greeting()

=> 결과는 =>

```
/Users/imac/PycharmProjects/untitle
Whee!
Whee!
Hello World!
Hello World!
Creating greeting
Creating greeting

Process finished with exit code 0
```

=> ?

Where is “Hi Adam” => ???

To get to our expectation, It should be like this

```python
def do_twice(func):
    def wrapper(*args):
        func(*args)
        return func(*args)
    return wrapper

@do_twice
def say_whee():
    print("Whee!")


@do_twice
def greet(name):
    print(f"Hello {name}")


say_whee()
greet("World!")


@do_twice
def return_greeting(name):
    print("Creating greeting")
    return f"Hi {name}"


a = return_greeting("Adam")
print(a)
```

=> In upper wrapper function, we didn't return anything. That means return value doesn't even exist.

It should be like this

+ => And In this case, if we want to actually see the return value we should print that

@property, .setter, .deleter

- **@property(decorator)**: 속성으로서 지정해버린다는 의미

```
class Employee:

    def __init__(self, first, last):
        self.first = first
        self.last = last

    @property
    def email(self):
        return '{} {}@email.com'.format(self.first, self.last)

    @property
    def fullname(self):
        return '{} {}'.format(self.first, self.last)
```

```
emp_1 = Employee('John', 'Smith')
print(1, emp_1.fullname)
```

```
1 John Smith
```

보면 @property를 달아서 사용하니 함수를 그냥 property 값처럼 사용하는 것을 볼 수 있다.

하지만 그 property값을 그냥 간단하게 할당하는 행위로 바꿀 수는 없다.

```
emp_1.fullname = 'Corey Schafer'
```

아래와 같은 에러가 뜬다. AttributeError: Can't set attribute

```
Traceback (most recent call last):
  File "/Users/coreyschafer/Projects/Demos/Python/OOP/oop.py", line 18, in <module>
    emp_1.fullname = 'Corey Schafer'
AttributeError: can't set attribute
[Finished in 0.0s with exit code 1]
[cmd: ['/Users/coreyschafer/anaconda/bin/python', '-u', '/Users/coreyschafer/Projects/Demos/Python/OOP/
oop.py']]
```

=> fullname의 정의를 보면 fullname은 인스턴스인 self에 대해 self.first와 self.last에 접근해서 만들었던 것인데 그것은 이전에 만들어질 당시 있었던 원래의 것으로 정해져 있기 때문이다. 그것 자체를 바꾸는 게 우리의 목적인데 새로운 인자를 가지고 인스턴스를 재생성할 수 도 없는 노릇이고,,(그것은 또하나의 독립적인 인스턴스를 만드는 것이지 기존의 인스턴스를 수정하는게 아니기때문)

=> 방법은 **setter** 를 사용하면 된다.(setter => 할당에 대한 것)

```
    @fullname.setter
    def fullname(self, name):
        first, last = name.split(' ')
        self.first = first
        self.last = last
```

=>함수의 이름을 fullname 이라고 맞춰주지 않아도 호출할때 여기서 정의한 이름으로만 해주면 사용할 수 있는데 그렇게 되면 이상하지 않은가? fullname에 대한 새로운 할당을 하는데 fullname이 아닌 다른것(여기서 정의한 이름)에 하는 것처럼 보여지니 말이다.

위와 같이 정의하고

```
emp_1.fullname = 'Corey Schafer'
```

다시 실행해보면 , 아래의 것에 대해서 다음과 같은 결과가 나오게 된다.

```
print(emp_1.first)
print(emp_1.email)
print(emp_1.fullname)
```

=>

```
Corey
Corey Schafer@email.com
Corey Schafer
```

=> Deleter는 delete => del에 관한것

```
    @fullname.deleter
    def fullname(self):
        print('Delete Name!')
        self.first = None
        self.last = None
```

```
del emp_1.fullname
```

```
print(emp_1.first)
print(emp_1.email)
print(emp_1.fullname)
```

=>

```
Delete Name!
None
None None@email.com
None None
```

An underscore infront of name

“_name” => one of naming Convention

: 해당의 것이 외부로부터 접근 되어지지 않게 하고 싶을 때

Good Inquiry and Answer for that from StackOverflow

https://stackoverflow.com/questions/1641219/does-python-have-private-variables-in-classes

Inquiry)

I'm coming from the Java world and reading Bruce Eckels' *Python 3 Patterns, Recipes and Idioms*.

While reading about classes, it goes on to say that in Python there is no need to declare instance variables. You just use them in the constructor, and boom, they are there.

So for example:

```
class Simple:
    def __init__(self, s):
        print("inside the simple constructor")
        self.s = s

    def show(self):
        print(self.s)

    def showMsg(self, msg):
        print(msg + ':', self.show())
```

If that’s true, then any object of class `Simple` can just change the value of variable `s` outside of the class.

For example:

```
if __name__ == "__main__":
    x = Simple("constructor argument")
    x.s = "test15" # this changes the value
    x.show()
    x.showMsg("A message")
```

In Java, we have been taught about public/private/protected variables. Those keywords make sense because at times you want variables in a class to which no one outside the class has access to.

Why is that not required in Python?

Answer)

It's cultural. In Python, you don't write to other classes' instance or class variables. In Java, nothing prevents you from doing the same if you *really* want to - after all, you can always edit the source of the class itself to achieve the same effect. Python drops that pretence of security and encourages programmers to be responsible. In practice, this works very nicely.

If you want to emulate private variables for some reason, you can always use the `__` prefix from PEP 8. Python mangles the names of variables like `__foo` so that they're not easily visible to code outside the class that contains them (although you *can* get around it if you're determined enough, just like you *can* get around Java's protections if you work at it).

By the same convention, the `_` prefix means **stay away even if you're not technically prevented from doing so**. You don't play around with another class's variables that look like `__foo` or `_bar`.

By Myungwoo Ko, 18.12.19