

파이썬 3대장을 만나보자

데커레이터 비동기 메타프로그래밍

박승규

안녕하세요. 박승규입니다



카카오페이지에서 운영툴을 만들고 있습니다
<= 이건 아내가 그려준 그림입니다



파이썬 3대장을 만나보자



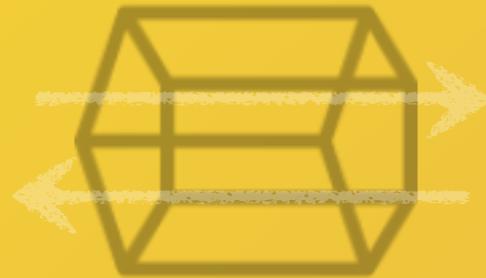
파이썬 3대장을 만나보자

정말 만나보고만 옵시다



@decorator

데커레이터



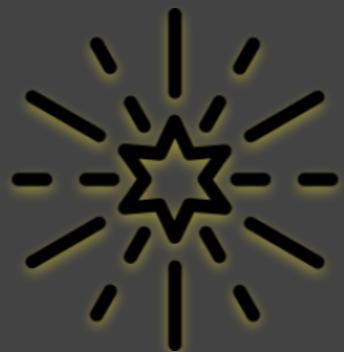
async

비동기

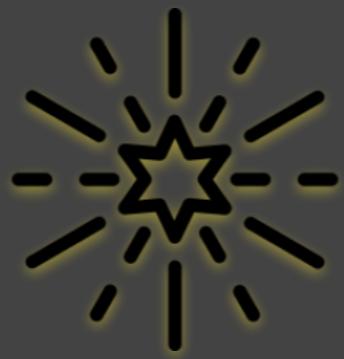


meta
programming

메타 프로그래밍

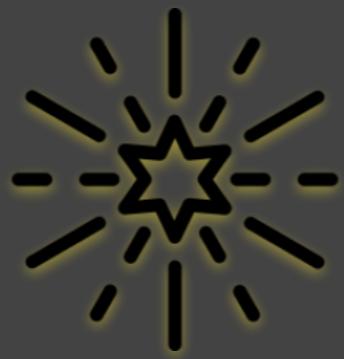


데커레이터란?



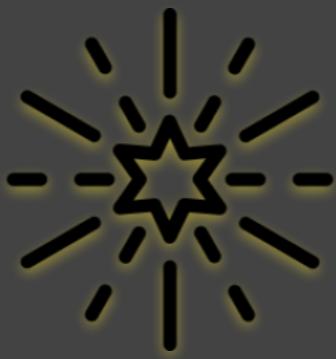
데커레이터란?

PEP 318 - 함수 메서드 및 데커레이터



데커레이터란?

함수에 '표시'를 해서 함수의 작동을 개선할 수 있게 해주는 것



데커레이터를 사용하면?

- 클로저를 알아야 한다
- 클로저를 이해하면 nonlocal 키워드를 어디에 쓸지 알게 된다



어디까지 알아볼까?

- 파이썬이 데커레이터 구문을 어떻게 평가하는지
- 파이썬은 어떤 변수가 로컬 변수인지 판단하는가
- 클로저는 어떻게 동작하나
- 잘 작동하는 데커레이터 만들기
- 매개변수화 된 데커레이터 만들기

데커레이터 기본지식

- 데커레이터는 다른 함수를 인수로 받는 callable 이다
- 데커레이터는 데커레이트된 함수에 어떤 처리를 수행하거나
함수를 반환하거나 다른 함수나 콜러블 객체로 대체한다.

데커레이터 기본지식

- 데커레이터는 다른 함수를 인수로 받는 callable 이다
- 데커레이터는 데커레이트된 함수에 어떤 처리를 수행하거나 함수를 반환하거나 다른 함수나 콜러블 객체로 대체한다.

저게 무슨말일까요?

데커레이터 기본지식

callable 쉽게 얘기해서
()로 실행할 수 있는 녀석입니다
클래스나 함수가 callable에 해당합니다.

- 데커레이터는 다른 함수를 인수로 받는 callable이다
- 데커레이터는 데커레이트된 함수에 어떤 처리를 수행하거나
함수를 반환하거나 다른 함수나 콜러블 객체로 대체한다.

```
def decorator(fun):  
    pass
```

데커레이터 기본지식

- 데커레이터는 다른 함수를 인수로 받는 callable 이다
- 데커레이터는 **데커레이트된 함수에 어떤 처리를 수행하거나**
함수를 반환하거나 다른 함수나 콜러블 객체로 대체한다.

```
def decorator(fun):  
    def decorated():  
        # 어떤 처리  
        result = fun()  
        return result  
  
    return decorated # 함수를 반환, 혹은 대체
```

데커레이터 기본지식

- 데커레이터는 다른 함수를 인수로 받는 callable 이다
- 데커레이터는 데커레이트된 함수에 어떤 처리를 수행하거나 함수를 반환하거나 다른 함수나 콜러블 객체로 대체한다.

```
def func():
    print('hello')

def decorator(fun):
    def inner_deco():
        # before
        return fun()
        # after

    return inner_deco

fun2 =
decorator(func)
```

@<func_name> 으로 장식을 해줍니다.



```
@decorator
def fn2():
    print("helooooo")
```

가장간단한 데커레이터

```
def decorate(func):
    def decorated():
        print('==' * 20)
        print('before')
        func()
        print('after')
    return decorated

@decorate
def target():
    print('target 함수')

target()

## output
=====
before
target 함수
after
=====
```

데커레이터는 아니지만 같은 동작

```
def target2():
    print('target2 함수 실행함')
```

```
target2 = decorate(target2)
target2()
```

```
## output
"""
=====
```

```
before
target2 함수 실행함
after
"""
=====
```

파이썬은 데커레이터를 언제 실행할까?

파이썬은 데커레이터를 언제 실행할까?

모듈을 로딩하는 시점 - 즉 import time

데커레이터는 모듈을 로딩하는 시점에 실행한다

```
registry = []

def register(func):
    print(f'레지스트리에 등록! {func}')
    registry.append(func)
    return func

@register
def f1():
    print('f1 실행')

@register
def f2():
    print('f2 실행')

@register
def f3():
    print('f3 실행')

def main():
    print('=====')
    print(f'레지스트리에 등록된거 보실? {registry}')
    f1()
    f2()
    f3()

if __name__ == '__main__':
    main()
```

데커레이터는 모듈을 로딩하는 시점에 실행한다

```
레지스트리에 등록! <function f1 at 0x10cb602f0>
```

```
레지스트리에 등록! <function f2 at 0x10cb60378>
```

```
레지스트리에 등록! <function f3 at 0x10cb60400>
```

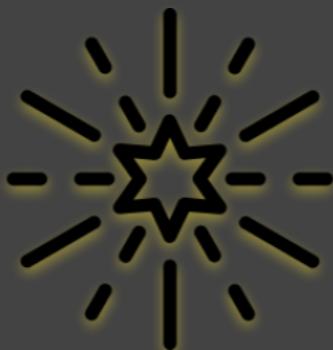
```
=====
```

```
레지스트리에 등록된거 보실? [<function f1 at 0x10cb602f0>, <function f2 at 0x10cb60378>, <function f3 at 0x10cb60400>]
```

```
f1 실행
```

```
f2 실행
```

```
f3 실행
```



어디까지 알아볼까?

- 파이썬이 데커레이터 구문을 어떻게 평가하는지
- 파이썬은 어떤 변수가 로컬 변수인지 판단하는가
- 클로저는 어떻게 동작하나
- 잘 작동하는 데커레이터 만들기
- 매개변수화 된 데커레이터 만들기

파이썬 함수 범위 규칙

```
# 변수 범위 규칙
def f1(a):
    print(a)
    print(b)
```

```
# 에러남
f1(3)
```

```
# 성공
b = 6
f1(3)
```

파이썬 함수 범위 규칙2

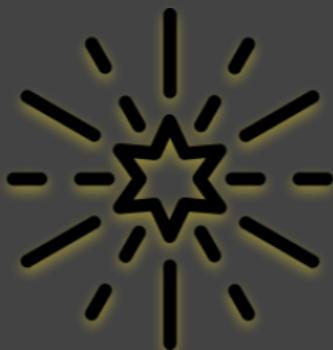
```
# 함수 본체 안에서 값을 할당하기 때문에 지역변수가 되는 b  
b = 6  
  
def f2(a):  
  
    print('a = ', a)  
    print('b = ', b)  
    b = 10  
  
f2(1)
```

파이썬 함수 범위 규칙2

```
# 함수 본체 안에서 값을 할당하기 때문에 지역변수가 되는 b
b = 6

def f2(a):
    global b
    print('a = ', a)
    print('b = ', b)
    b = 10

f2(1)
```



어디까지 알아볼까?

- 파이썬이 데커레이터 구문을 어떻게 평가하는지
- 파이썬은 어떤 변수가 로컬 변수인지 판단하는가
- 클로저는 어떻게 동작하나
- 잘 작동하는 데커레이터 만들기
- 매개변수화 된 데커레이터 만들기

클로저란?

자유 변수에 대한 바인딩을 유지하는 함수이다.

클로저란?

자유 변수에 대한 바인딩을 유지하는 함수이다.

코드블럭에서 사용되고 있지만, 선언은 되어있지 않은 변수를 말한다.

```
# x는 글로벌 변수이다.  
# foo 함수 내에서 x는 자유 변수이다.  
x = 1  
def foo():  
    print(x)
```

```
# outer_func는 지역변수 n이 있다.  
# inner_func에는 지역변수 m과 n이 있다.  
def outer_func():  
    n = 1  
    def inner_func():  
        m = 2  
        print(n)  
        print(m)  
  
    return inner_func  
func = outer_func()  
print(func.__code__.co_freevars)
```

클로저 연습1

```
# 아래와 같이 동작하는 avg 함수를 만들어보자
```

```
>>> avg(10)
```

```
10.0
```

```
>>> avg(11)
```

```
10.5
```

```
>>> avg(12)
```

```
11.0
```

클로저 연습1

```
class Avg():
    def __init__(self):
        self.nums = []

    def __call__(self, value):
        self.nums.append(value)
        total = sum(self.nums)
        return total / len(self.nums)

avg = Avg()

print(avg(10))
print(avg(11))
print(avg(12))
```

클로저 연습1

```
# 고계함수를 사용한 버전
def make_avg():
    arr = []

    def func(value):
        arr.append(value)
        total = sum(arr)
        return total / len(arr)
    return func

avg = make_avg()

print(avg(10))
print(avg(11))
print(avg(12))
```

클로저 연습1

```
# 고계함수를 사용한 버전
def make_avg():
    arr = []

    def func(value):
        arr.append(value)
        total = sum(arr)
        return total / len(arr)
    return func
```

```
avg = make_avg()
```

클로저를 찾아 봅시다!

```
print(avg(10))
print(avg(11))
print(avg(12))
```

클로저 연습1

```
# 고계함수를 사용한 버전
```

```
def make_avg():
```

```
    arr = []
```

```
    def func(value):
```

```
        arr.append(value)
```

```
        total = sum(arr)
```

```
        return total / len(arr)
```

함수

```
    return func
```

클로저

자유변수 arr

```
avg = make_avg()
```

```
print(avg(10))
```

```
print(avg(11))
```

```
print(avg(12))
```

클로저 연습1

```
# 고계함수를 사용한 버전
```

```
def make_avg():
    arr = []

    def func(value):
        arr.append(value)
        total = sum(arr)
        return total / len(arr)

    return func
```

함수

클로저

자유변수 arr

```
avg = make_avg()
```

이력을 저장하지 않는
make_avg2 를 만들어봅시다

```
print(avg(10))
print(avg(11))
print(avg(12))
```

클로저 연습2

```
def make_avg2():
    count = 0
    total = 0

    def func(value):
        count += 1
        total += value
        return total / count

    return func
```

```
avg = make_avg2()
```

잘 실행되나요?

```
print(avg(10))
print(avg(11))
print(avg(12))
```

클로저 연습2

```
def make_avg2():
    count = 0
    total = 0

    def func(value):
        count += 1
        total += value
        return total / count

    return func
```

```
avg = make_avg2()
```

```
print(avg(10))
print(avg(11))
print(avg(12))
```

count += 1 은

실제로는 count = count + 1 이므로
count 를 다시 할당하게 됩니다.

함수에서 값을 할당하면 **지역변수**가 됩니다.

클로저 연습2

```
def make_avg2():
    count = 0
    total = 0

    def func(value):
        count += 1
        total += value
        return total / count

    return func
```

```
avg = make_avg2()
```

```
print(avg(10))
print(avg(11))
print(avg(12))
```

이런경우에 쓰라고 python에서는
nolocal 키워드를 만들어 두었습니다.
count와 total을 nonlocal로 지정해줍시다

클로저 연습2

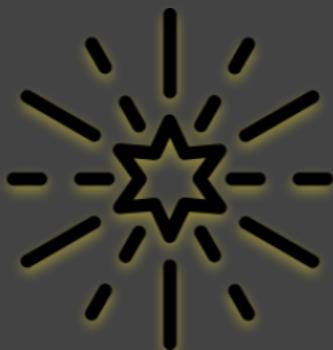
```
def make_avg2():
    count = 0
    total = 0

    def func(value):
        nonlocal count, total
        count += 1
        total += value
        return total / count

    return func

avg = make_avg2()

print(avg(10))
print(avg(11))
print(avg(12))
```



어디까지 알아볼까?

- 파이썬이 데커레이터 구문을 어떻게 평가하는지
- 파이썬은 어떤 변수가 로컬 변수인지 판단하는가
- 클로저는 어떻게 동작하나
- 잘 작동하는 데커레이터 만들기
- 매개변수화 된 데커레이터 만들기

간단한 데커레이터 만들기 | 1 clock

```
import time
def clock(func):
    def clocked(*args):
        before = time.perf_counter()
        result = func(*args)
        elapsed_time = time.perf_counter() - before
        arg_str = ', '.join(repr(arg) for arg in args)
        print(f'{elapsed_time:.8f} {func.__name__} {arg_str} {result}')
        return result
    return clocked
```

```
@clock
def snooze(seconds):
    time.sleep(seconds)
```

```
@clock
def factorial(n):
    return 1 if n < 2 else n * factorial(n -1)
```

```
print('*' * 40, 'call snooze(.123)')
print("func snooze :", snooze)
snooze(.123)
```

```
print('*' * 40, 'call factorial(10)')
print("func factorial :", factorial)
print("10! = ", factorial(10))
```

간단한 데커레이터 만들기 | 1 clock 결과

```
***** call snooze(.123)
func snooze : <function clock.<locals>.clocked at 0x10c830e18>
0.12344381 snooze 0.123 None
***** call factorial(10)
func factorial : <function clock.<locals>.clocked at 0x10c830f28>
0.00000038 factorial 1 1
0.00000927 factorial 2 2
0.00001461 factorial 3 6
0.00001916 factorial 4 24
0.00002410 factorial 5 120
0.00002882 factorial 6 720
0.00003372 factorial 7 5040
0.00003832 factorial 8 40320
0.00004301 factorial 9 362880
0.00004957 factorial 10 3628800
10! = 3628800
```

간단한 데커레이터 만들기 | 1 clock

```
import time
def clock(func):
    def clocked(*args):
        before = time.perf_counter()
        result = func(*args)
        elapsed_time = time.perf_counter() - before
        arg_str = ', '.join(repr(arg) for arg in args)
        print(f'{elapsed_time:.8f} {func.__name__} {arg_str} {result}')
        return result
    return clocked
```

```
@clock
def snooze(seconds):
    time.sleep(seconds)
```

단점이 몇가지 있습니다 뭘까요?

```
@clock
def factorial(n):
    return 1 if n < 2 else n * factorial(n -1)
```

```
print("*" * 40, 'call snooze(.123)')
print("func snooze :", snooze)
snooze(.123)
```

```
print("*" * 40, 'call factorial(10)')
print("func factorial :", factorial)
print("10! = ", factorial(10))
```

간단한 데커레이터 만들기 | 1 clock

```
import time
def clock(func):
    def clocked(*args):
        before = time.perf_counter()
        result = func(*args)
        elapsed_time = time.perf_counter() - before
        arg_str = ','.join(repr(arg) for arg in args)
        print(f'{elapsed_time:.8f} {func.__name__} {arg_str} {result}')
        return result
    return clocked
```

```
@clock
def snooze(seconds):
    time.sleep(seconds)
```

키워드 인수(kwargs)를 지원하지 않는다.

```
@clock
def factorial(n):
    return 1 if n < 2 else n * factorial(n -1)
```

```
print("*" * 40, 'call snooze(.123)')
print("func snooze :", snooze)
snooze(.123)
```

```
print("*" * 40, 'call factorial(10)')
print("func factorial :", factorial)
print("10! = ", factorial(10))
```

간단한 데커레이터 만들기 1 clock 결과

```
***** call snooze(.123)
func snooze : <function clock.<locals>.clocked at 0x10c830e18>
0.12344381 snooze 0.123 None
***** call factorial(10)
func factorial : <function clock.<locals>.clocked at 0x10c830f28>
0.00000038 factorial 1 1
0.00000927 factorial 2 2
0.00001461 factorial 3 6
0.00001916 factorial 4 24
0.00002410 factorial 5 120
0.00002882 factorial 6 720
0.00003372 factorial 7 5040
0.00003832 factorial 8 40320
0.00004301 factorial 9 362880
0.00004957 factorial 10 3628800
10! = 3628800
```

데커레이트된 함수의
__name__ 과
__doc__ 속성을 가린다

간단한 데커레이터 만들기 1 clock 결과

```
***** call snooze(.123)
func snooze : <function clock.<locals>.clocked at 0x10c830e18>
0.12344381 snooze 0.123 None
***** call factorial(10)
func factorial : <function clock.<locals>.clocked at 0x10c830f28>
0.00000038 factorial 1 1
0.00000927 factorial 2 2
0.00001461 factorial 3 6
0.00001916 factorial 4 24
0.00002410 factorial 5 120
0.00002882 factorial 6 720
0.00003372 factorial 7 5040
0.00003832 factorial 8 40320
0.00004301 factorial 9 362880
0.00004957 factorial 10 3628800
10! = 3628800
```

데커레이트된 함수의
__name__ 과
__doc__ 속성을 가린다

@functools.wraps를 쓰면 됩니다

간단한 데커레이터 만들기 | 1 clock 수정

```
import time
import functools
def clock(func):
    @functools.wraps(func)
    def clocked(*args, **kwargs):
        before = time.perf_counter()
        result = func(*args, **kwargs)
        elapsed_time = time.perf_counter() - before
        arg_str = ', '.join(repr(arg) for arg in args)
        if kwargs:
            print('kwargs : ', **kwargs)
        print(f'{elapsed_time:.8f} {func.__name__} {arg_str} {result}')
        return result
    return clocked

@clock
def snooze(seconds):
    time.sleep(seconds)

@clock
def factorial(n):
    return 1 if n < 2 else n * factorial(n - 1)

print('*' * 40, 'call snooze(.123)')
print("func snooze :", snooze)
snooze(.123)

print('*' * 40, 'call factorial(10)')
print("func factorial :", factorial)
print("10! = ", factorial(10))
```

간단한 데커레이터 만들기 1 clock 수정후 결과

```
***** call snooze(.123)
func snooze : <function snooze at 0x100a79ae8>
0.12674616 snooze 0.123 None
***** call factorial(10)
func factorial : <function factorial at 0x100a79bf8>
0.00000044 factorial 1 1
0.00000998 factorial 2 2
0.00001533 factorial 3 6
0.00002012 factorial 4 24
0.00002527 factorial 5 120
0.00003171 factorial 6 720
0.00003678 factorial 7 5040
0.00004151 factorial 8 40320
0.00005142 factorial 9 362880
0.00005870 factorial 10 3628800
10! = 3628800
```

데커레이트된 함수의
name 이 잘 나오고 있다

데이터레이터 템플릿

아래의 형태를 데커레이터 만들때의 템플릿으로 생각하면 좋다.

```
def decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        # 전처리
        result = func(*args, **kwargs)
        # 후처리
        return result
    return wrapper
```

데커레이터 템플릿

아래의 형태를 데커레이터 만들때의 템플릿으로 생각하면 좋다.

```
def decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        # 전처리
        result = func(*args, **kwargs)
        # 후처리
        return result
    return wrapper
```

데커레이터에 파라메터를 넣고 싶다면?

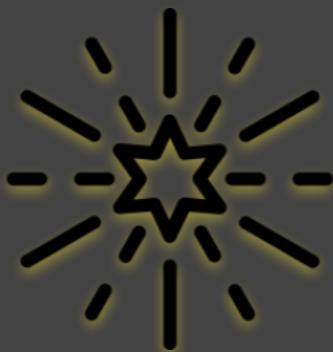
데커레이터 템플릿

```
import functools

# 파라메터가 있는 데커레이터를 위한 템플릿
def decorator_with_param(param):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            # 전처리
            result = func(*args, **kwargs)
            # 후처리
            return result

    return wrapper

return decorator
```



어디까지 알아볼까?

- 파이썬이 데커레이터 구문을 어떻게 평가하는지
- 파이썬은 어떤 변수가 로컬 변수인지 판단하는가
- 클로저는 어떻게 동작하나
- 잘 작동하는 데커레이터 만들기
- 매개변수화 된 데커레이터 만들기

파라메터가 있는 데커레이터

```
import functools
def title(symbol, len=20):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            print(symbol * len)
            result = func(*args, **kwargs)
            print(symbol * len)
            return result
        return wrapper
    return decorator
```

```
@title('*', 15)
def print_name(name):
    print(name)
    return name
```

```
@title('-', 10)
def print_company(company):
    print(company)
    return company
```

```
print_name("seungkyoo park")
print_company("kakaopage")
```

파라메터가 있는 데커레이터2

```
import time
import functools

DEFAULT_FORMAT= '[ELAPSED_TIME : {elapsed_time:.8f}] {name} {_args} -> {result}'

def clock(format=DEFAULT_FORMAT):
    def decorator(func):
        @functools.wraps(func)
        def clocked(*args, **kwargs):
            before = time.perf_counter()
            result = func(*args, **kwargs)
            _args = ','.join(repr(arg) for arg in args)
            elapsed_time = time.perf_counter() - before
            name = func.__name__
            print(format.format(**locals()))
            return result
        return clocked
    return decorator
```

```
@clock()
def snooze(seconds):
    time.sleep(seconds)

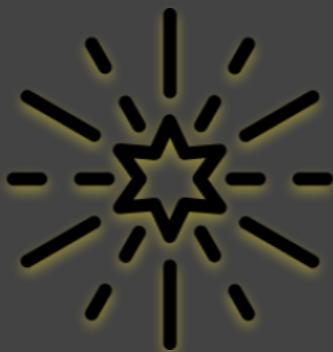
@clock('{name} calls spend {elapsed_time:.8f} s')
def factorial(n):
    return 1 if n < 2 else n * factorial(n - 1)

print('*' * 40, 'call snooze(.123)')
snooze(.123)

print('*' * 40, 'call factorial(10)')
print("10! = ", factorial(10))
```

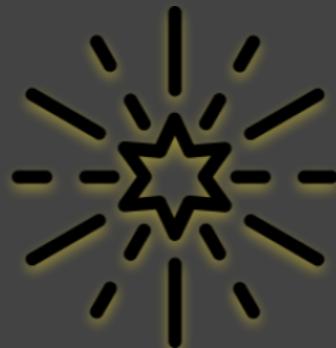
파라메터가 있는 데커레이터2_결과

```
***** call snooze(.123)
[ELAPSED_TIME : 0.12700218] snooze 0.123 -> None
***** call factorial(10)
[factorial] calls spend 0.00000261 s
[factorial] calls spend 0.00001560 s
[factorial] calls spend 0.00002230 s
[factorial] calls spend 0.00002794 s
[factorial] calls spend 0.00003375 s
[factorial] calls spend 0.00004102 s
[factorial] calls spend 0.00004659 s
[factorial] calls spend 0.00005249 s
[factorial] calls spend 0.00006240 s
[factorial] calls spend 0.00006959 s
10! = 3628800
```



어디까지 알아볼까?

- 파이썬이 데커레이터 구문을 어떻게 평가하는지
- 파이썬은 어떤 변수가 로컬 변수인지 판단하는가
- 클로저는 어떻게 동작하나
- 잘 작동하는 데커레이터 만들기
- 매개변수화 된 데커레이터 만들기



더 알아 보기

- 클래스 데커레이터
- 클래스 데커레이터로 싱글톤 만들기
- 파이썬에 내장된 좋은 데커레이터들
 - lru_cache, dispatch 등
 - 데커레이터 내부에서 예외 처리
 - 실제 업무에 자신만의 데커레이터 만들어서 사용하기



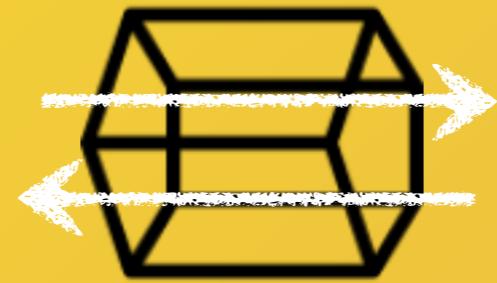
참고자료

- 전문가를 위한 파이썬 7장 데커레이터와 클로저
- [real python](#)
- [python wiki : Python Decorator Library](#)



@decorator

데코레이터



async

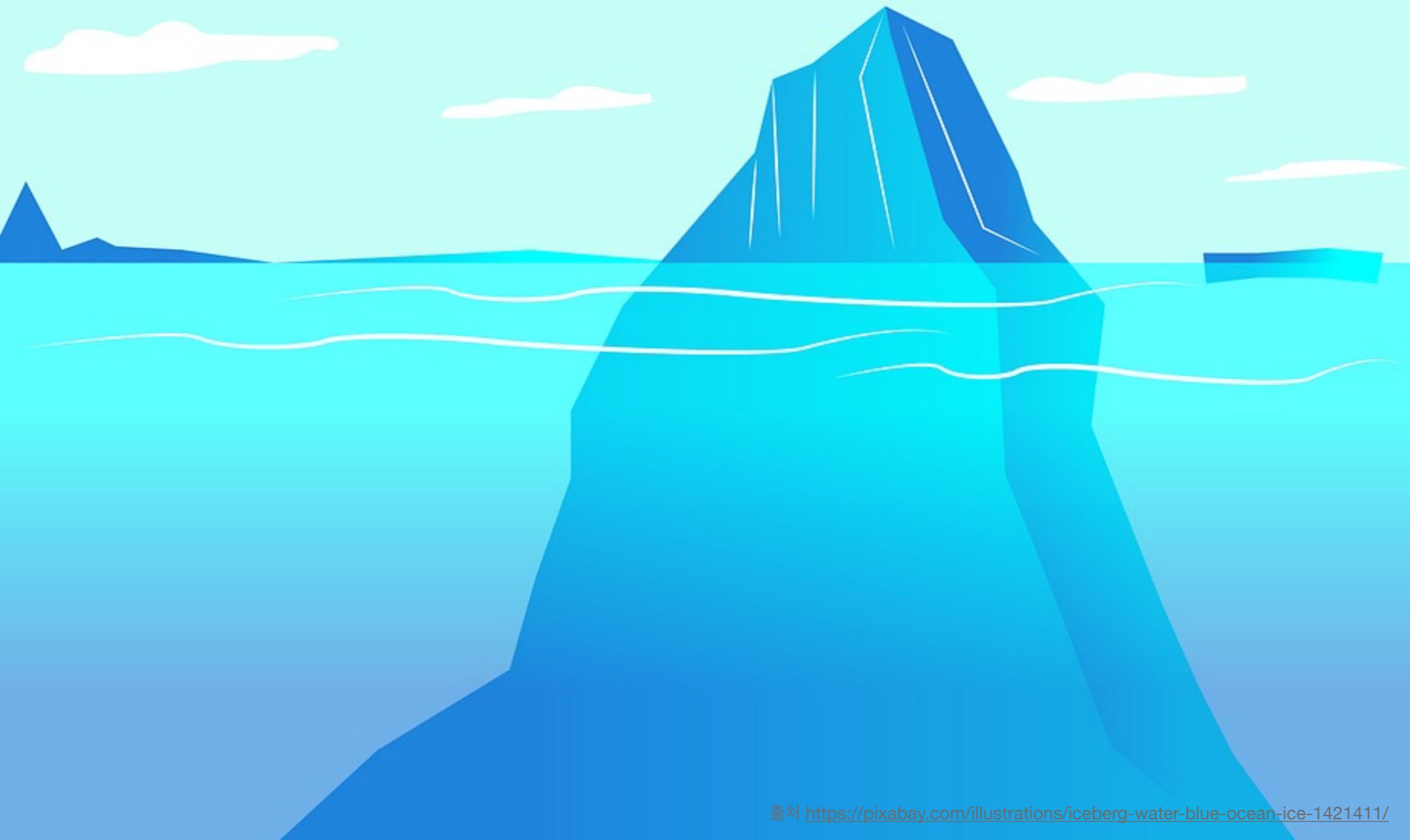
비동기



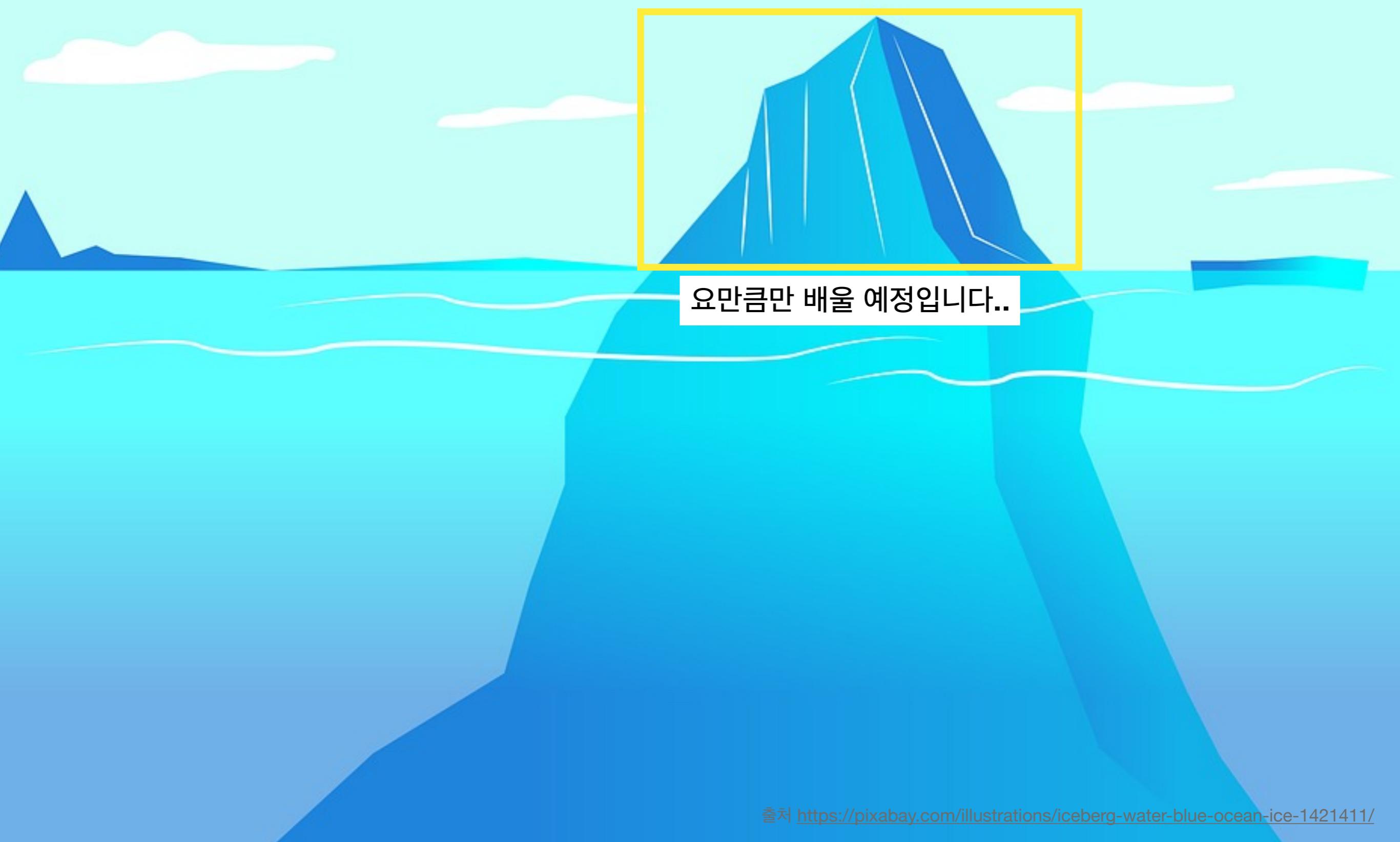
meta
programming

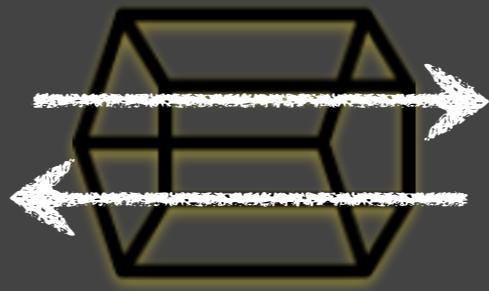
메타 프로그래밍

빙산의 일각



async의 일각





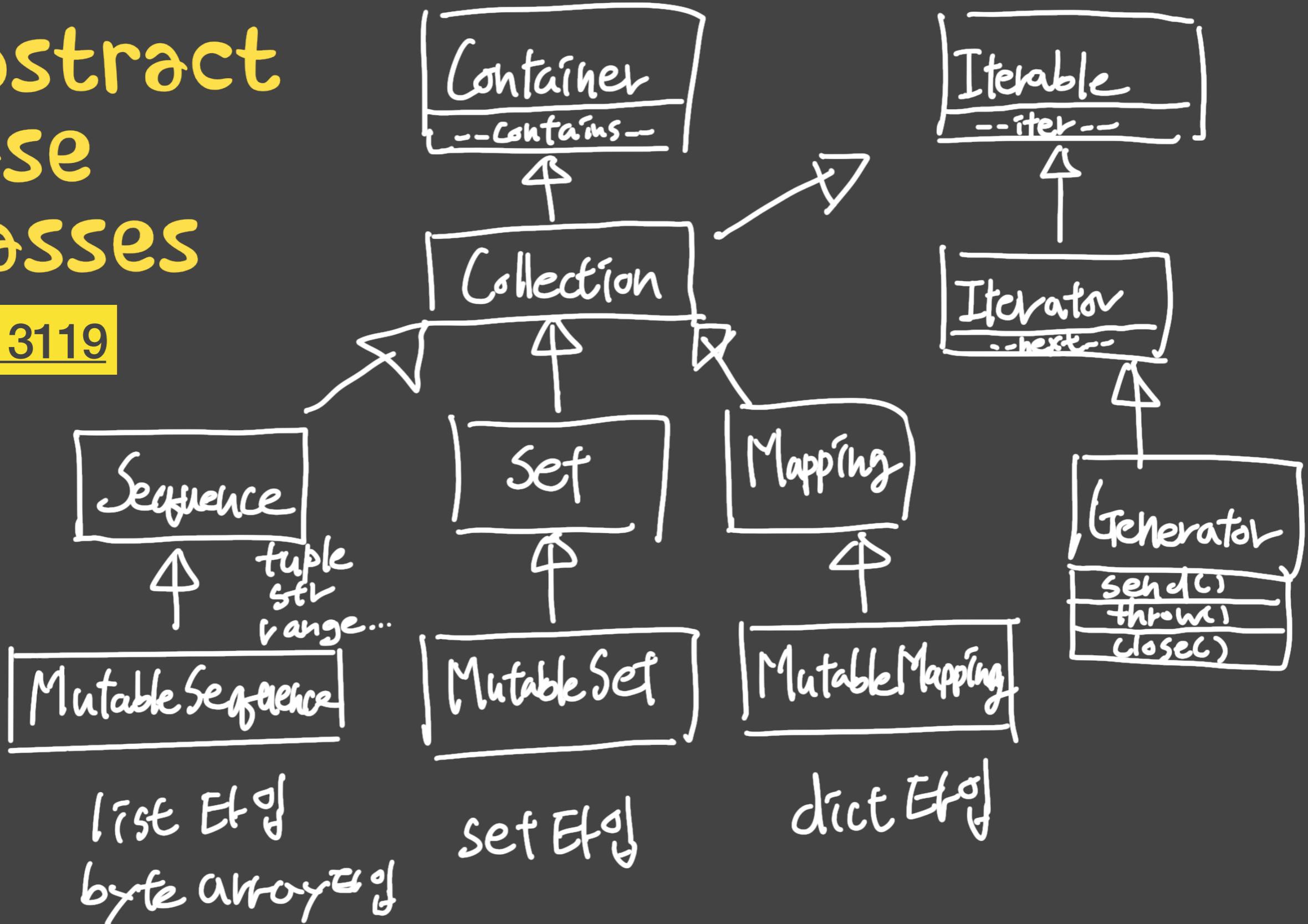
async로 가는 여정

python 3.7 이 필요해요

- iterable, iterator, generator, coroutine
- yield from
- yield from 을 사용해서 흐름제어 하기
- yield from 을 async await 로 변경하기
- async await 예제

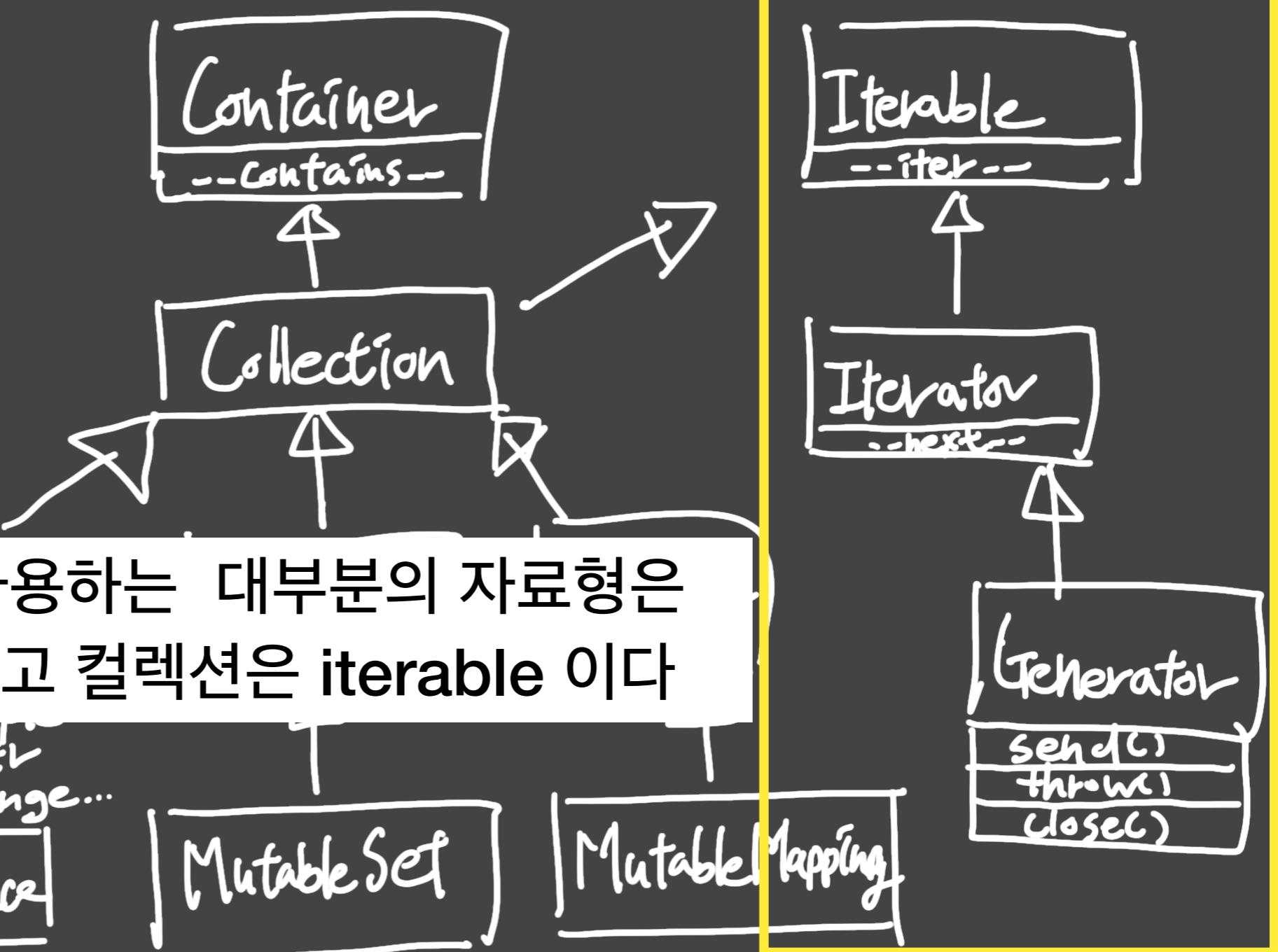
Abstract Base Classes

PEP 3119



Abstract Base Classes

PEP 3119

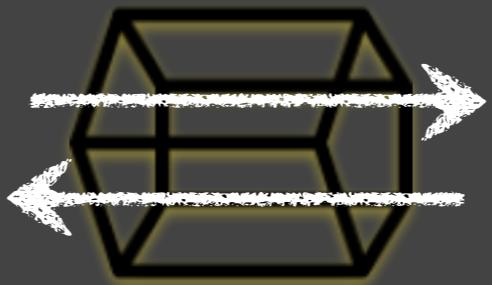


우리가 사용하는 대부분의 자료형은
컬렉션이고 컬렉션은 iterable 이다

list 타입
bytearray 타입

set 타입

dict 타입



async로 가는 여정

python 3.7 이 필요해요

- **iterable, iterator, generator**
- generator 를 사용하는 3가지 방법
- asyncio

iterable, iterator, generator

```
# async 를 공부하자고 하는데 좀 쌩뚱맞을 수 있지만  
# 간단히 1부터 5까지 표시해주는 for loop 를 만들어봅시다.  
# 나중에 이야기 하겠지만, 이 간단한 아이디어에서 async 가 시작되기 때문에 그렇습니다.
```

```
nums = [1, 2, 3, 4, 5]
```

```
for num in nums:  
    print(num)
```

iterable, iterator, generator

```
# for num in nums 에서는 무슨일이 일어날까요?  
# 실제로는 아래와 같은 형태로 구현되어 있습니다.  
# iter 함수로 iterator 를 만들고  
# next 를 사용해서 하나씩 꺼내오게 됩니다.  
# 더이상 꺼내올 것이 없으면 StopIteration 이 나면서 종료됩니다.  
iter_obj = iter(nums)  
while True:  
    try:  
        el = next(iter_obj)  
        print(el)  
    except StopIteration:  
        break
```

iterable, iterator, generator

```
# iter, next 를 직접 호출해서  
# 잘 동작하는지 실행해봅시다.
```

```
it = iter(nums)  
print(next(it))  
print(next(it))  
print(next(it))  
print(next(it))  
print(next(it))
```

```
# 여기서 중단됨  
print(next(it))
```

```
1  
2  
3  
4  
5
```

```
Traceback (most recent call last):
```

```
  File "/Users/gyus/dev/python/pycon2019/async_example/iterator1.py", line 36, in <  
    module>  
        print(next(it))  
StopIteration
```

iterable, iterator, generator

```
# iter 함수를 사용해서 iterator 로 만드려면 iterable 이어야 합니다.  
# 그림에서 봤던대로 iterable 은 __iter__() 메서드를 구현해주면 됩니다  
# for loop 에서 사용할 수 있게 iterator 가 되려면 __next__() 함수도 구현해주어야 합니다.  
# 루프를 돌때마다 1씩 증가하고 5까지만 세어주는  
# Counter5 클래스를 만들고 __iter__(), __next__() 함수를 구현해 줍시다.  
# __next__ 에서 리턴해줄 것이 없으면 StopIteration 이 나도록 해주면 for loop 에서 빠져나오게 됩니다.
```

```
class Counter5:  
    LIMIT = 5  
  
    def __init__(self):  
        self.num = 1  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        val = self.num  
        self.num += 1  
        if self.LIMIT < val:  
            raise StopIteration  
        return val
```

```
counter = Counter5()  
for n in counter:  
    print(n)
```

iterable, iterator, generator

```
# Counter 클래스는 아래와 같이 제너레이터로 변경할 수 있습니다.  
# yield 는 iterator 를 추상화 하기 위해 Python 2.2 에 추가되었습니다.
```

```
def gen_1_to_5():  
    yield 1  
    yield 2  
    yield 3  
    yield 4  
    yield 5
```

```
counter = gen_1_to_5()  
print(type(counter))  
print(next(counter))  
print(next(counter))  
print(next(counter))  
print(next(counter))  
print(next(counter))  
print(next(counter))
```

```
# StopIteration 발생  
# print(next(counter))
```

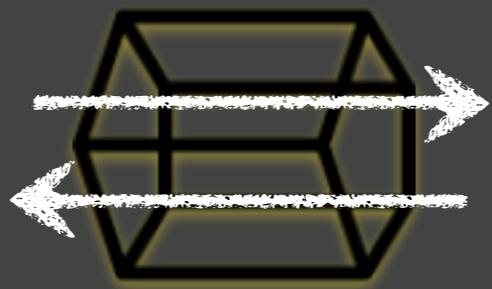
```
# 리스트로 바로 변경할 수 있습니다.  
print(list(count_to(5)))
```

iterable, iterator, generator

```
# generator 는 iterator 를 만들어내는 함수라고 일단 생각합시다.  
# 일일이 쓰는것은 프로그래머스럽지 않으니 아래와 같이 바꿔봅시다.
```

```
def count_to(n):  
    num = 1  
    while num <= n:  
        yield num  
        num += 1
```

```
counter = count_to(5)  
print(type(counter))  
print(list(counter))
```



async로 가는 여정

python 3.7 이 필요해요

- iterable, iterator, generator
- **generator** 를 사용하는 3가지 방법
- asyncio

제너레이터를 사용하는 3가지 방법

“풀” 스타일(iterator) -> 이미 해봤음

“푸시” 스타일(coroutine)

“작업” (asyncio task)

제너레이터를 사용하는 3가지 방법

“풀” 제너레이터를 사용하면 메모리를 점점

“푸시” 효율적으로 사용할 수 있습니다.

“작업” 실제로 그런지 한번 알아봅시다.

generator

제너레이터라는 것을 알아보긴 했지만, 제너레이터를 사용해야하는

이유는 무엇일까요?

```
import random
import os
import time
import psutil

process = psutil.Process(os.getpid())
titles = ["달빛조각사", "김비서가 왜그럴까", "롱리브더킹",
          "나 혼자만 레벨업", "독고", "묵향"]
expose = ["메인홈", "랭킹", "이벤트페이지", "뽑기권",
          "카테고리홈", "기다리면 무료", "오리지널"]

def memory_usage_mb():
    return process.memory_info().rss / 1000000
```

```
mem_before = memory_usage_mb()
print(f"Memory (Before): {mem_before:.2f}Mb")
```

```
def series_product_list(size):
    result = []
    for n in range(size):
        series = {
            "id": n,
            "title": random.choice(titles),
            "expose": random.choice(expose),
            "update_dt": time.time(),
        }
        result.append(series)
    return result
```

```
def series_product_gen(size):
    for n in range(size):
        series = {
            "id": n,
            "title": random.choice(titles),
            "expose": random.choice(expose),
            "update_dt": time.time(),
        }
        yield series
```

```
SIZE = 3000000
before = time.perf_counter()
# series_list = series_product_list(SIZE)
series_gen = series_product_gen(SIZE)
after = time.perf_counter()

mem_after = memory_usage_mb()
print(f"Memory (After): {mem_after:.2f}Mb")
print(f"Used Memory : {(mem_after - mem_before):.2f}Mb")
print(f"Elapsed Time : {(after - before):.2f}")
```

generator

```
# 리스트 사용시
```

```
Memory (Before): 10.47Mb  
Memory (After): 1002.76Mb  
Used Memory : 992.29Mb  
Elapsed Time : 6.25
```

```
# 제너레이터 사용시
```

```
Memory (Before): 10.45Mb  
Memory (After): 10.47Mb  
Used Memory : 0.02Mb  
Elapsed Time : 0.00
```

리스트(이터레이터)를 가져오는
메서드를 실행하는 경우는
제너레이터가 메모리 효율이 좋습니다

generator

```
# 제너레이터로 구구단도 만들어봅시다.  
def gugu_gen():  
    for x in range(1, 10):  
        for y in range(1, 10):  
            print(f"{x} * {y} = ", end="")  
            yield x * y  
  
gugu = gugu_gen()  
  
for result in gugu:  
    print(result)
```

제너레이터를 사용하는 3가지 방법

“풀” 스타일(iterator) -> 이미 해봤음

“푸시” 스타일(coroutine)

“작업” (asyncio task)

제너레이터를 사용하는 3가지 방법

- 파이썬 2.5(PEP 342)에 코루틴이 구현되었다.
- 제너레이터 객체에 `send()` 등의 메서드와 기능을 추가했다.
- 호출자와 제너레이터가 양방향으로 데이터를 교환할 수 있다.
- `throw()`로 코루틴에 예외를 전달 할 수 있다.
- `close()`를 사용해서 코루틴을 종료시킬 수 있다.

coroutine : pingpong

```
def coru1(param):
    while True:
        msg_from_caller = yield param
        print(msg_from_caller)

cr = coru1("ping")

print(next(cr))
cr.send("pong")
```

coroutine : pingpong

```
class HighNumberException(Exception):
    pass

def infinite_pingpong(param):
    while True:
        try:
            msg_from_caller = yield param
            print(msg_from_caller)
        except HighNumberException:
            print("숫자가 높아요")
            break
        except GeneratorExit:
            print("코루틴 종료! " + msg_from_caller)

def coru_throw():
    for x in range(10):
        coru = infinite_pingpong(f"pping {x}")
        if x > 1:
            coru.throw(HighNumberException)
        print(next(coru))
        coru.send(f"ppong {x}")

def coru_close():
    for x in range(100):
        coru = infinite_pingpong(f"pingping {x}")
        print(next(coru))
        coru.send(f"pongpong {x}")
        if x >= 3:
            coru.close()
# 종료시에는 모든 코루틴을 종료시킨다.
```

coroutine : 이동평균

```
# 코루틴으로 이동편균 구하기
# @coroutine 으로 코루틴 기동시키기
# 결과값을 받아보기
import functools

def coroutine(func):
    @functools.wraps(func)
    def wrapped(*args, **kwargs):
        gen = func(*args, **kwargs)
        next(gen)
        return gen
    return wrapped
```

```
@coroutine
def averager():
    sum = 0.0
    count = 0
    average = None
    while True:
        n = yield average
        if n is None:
            break
        sum += n
        count += 1
        average = sum / count
    return (count, average)

coro_avg = averager()
print(coro_avg.send(10))
print(coro_avg.send(20))
print(coro_avg.send(30))
# stop Iteration 이 발생! 하지만 결과값이 들어있다!
try:
    print(coro_avg.send(None))
except StopIteration as exc:
    result = exc.value

print(result)
```

제너레이터를 사용하는 3가지 방법

“풀” 스타일(iterator) -> 이미 해봤음

“푸시” 스타일(coroutine)

“작업” (yield from, asyncio)

yield from

```
# yield from 은 PEP380 에 서브제너레이터 문법을 위한 문법으로  
제안되었습니다.
```

```
# 간단한 사용법부터 알아봅시다 .
```

```
# 아래와 같이 for loop yield 가 두개 있는 함수가 있습니다.
```

```
# range(n, m) 부분을 제너레이터로 변경해 볼까요
```

```
def gen():  
    for i in range(10):  
        yield i  
    for j in range(10, 20):  
        yield j
```

yield from

```
# 아래와 같이 리팩토링이 가능하겠죠
```

```
def gen1():
    for i in range(10):
        yield i

def gen2():
    for j in range(10,20):
        yield j

def ggen():
    for i in gen1():
        yield i

    for j in gen2():
        yield j
```

yield from

```
# yield from 으로 쉽게 변경할 수 있습니다.  
# 즉 yield from 은  
for item in iterable:  
    yield item  
# 의 축약형으로 사용할 수 있습니다.  
  
def gggen():  
    yield from gen1()  
    yield from gen2()
```

yield from

```
# PEP380 에 나온 서브 제너레이터에 위임하기 구문에 대한 것도 알아봅시다.  
# 메인 제너레이터, 서브 제너레이터, 호출자 3가지에 대해서 알아야합니다.  
#  
# 호출자  
# - 대표 제너레이터를 호출하는 코드 주로 메인함수.  
# 메인 제너레이터  
# - yield from <coroutine> 표현식이 있는 함수.  
# 서브 제너레이터  
# - yield from <coroutine>에서 coroutine 역할을 담당하고 있음
```

yield from

```
# 서브제너레이터
def averager():
    print("averager")
    avg = 0.0
    total = 0
    count = 0
    while True:
        val = yield
        print("위임제너레이터에서 넘어온 값"
, val) if val is None:
            return avg, count
        total += val
        count += 1
        avg = total / count

# 메인 제너레이터
def delegating_gen(results):
    while True:
        result = yield from averager()
        results.append(result)

# 호출자
def main():
    ages = [21, 29, 33, 39]
    weights = [48, 63, 71, 68]
    iterables = [ages, weights]
    results = []
    for iter in iterables:
        gen = delegating_gen(results)
        next(gen)
        print(gen)
        for val in iter:
            print("main in val :", val)
            gen.send(val)
        gen.send(None)
    print("<최종 결과>", results)

main()
```

yield from

```
# output
averager
<generator object delegating_gen at 0x10b81aa98>
main in val : 21
위임제너레이터에서 넘어온 값 21
main in val : 29
위임제너레이터에서 넘어온 값 29
main in val : 33
위임제너레이터에서 넘어온 값 33
main in val : 39
위임제너레이터에서 넘어온 값 39
위임제너레이터에서 넘어온 값 None
averager
averager
<generator object delegating_gen at 0x10b893a98>
main in val : 48
위임제너레이터에서 넘어온 값 48
main in val : 63
위임제너레이터에서 넘어온 값 63
main in val : 71
위임제너레이터에서 넘어온 값 71
main in val : 68
위임제너레이터에서 넘어온 값 68
위임제너레이터에서 넘어온 값 None
averager
<최종 결과> [(30.5, 4), (62.5, 4)]
```

yield from

호출자

메인 제너레이터

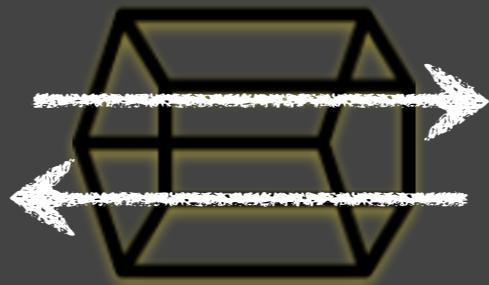
서브 제너레이터

```
gen = delegating_gen(results)  
gen.send(val)
```

```
yield from averager()
```

```
yield
```





async로 가는 여정

- iterable, iterator, generator
- generator 를 사용하는 3가지 방법
- **asyncio** 드디어!

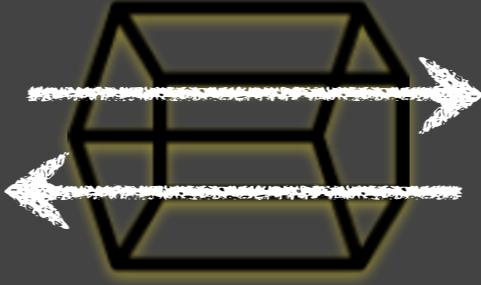
asyncio

- **concurrency vs parallelism**
- yield from coroutine vs yield from task
- yield from -> async await
- spinner 예제
- clock 예제
- 파일 다운로드 받기
- 파일 다운로드 여러개 받기

Concurrency vs. parallelism

- 동시성은 한번에 여러가지를 다루는 것에 대한 것이다.
- 병렬성은 한번에 여러가지를 실행하는 것에 대한 것이다.
- 같은 것은 아니지만, 관련이 있다.
- 동시성은 구조에 대한 것이고, 병렬성은 실행에 대한 것이다.
- 동시성은 병렬화에 대한 문제를 구조적 풀 수 있는 답을 제공해준다.

- rob pike -



asyncio

- concurrency vs parallelism
- **yield from coroutine vs yield from task**
- yield from -> async await
- spinner 예제
- clock 예제
- 파일 다운로드 받기
- 파일 다운로드 여러개 받기

yield from coroutine vs yield from task

```
import asyncio
import time

# 파이썬을 만든 귀도반 로썸은
# tulip 프로젝트를(asyncio) 소개하는 슬라이드에서 아래와 같이 적어두었습니다.
# Tasks vs coroutines
# res = yield from some_coroutine()
# res = yield from Task(some_coroutine())

# 태스크는 Future로 감싼 코루틴인데요.
# 파이썬의 asyncio.py 코드에 아래와 같이 되어 있기때문입니다.
class Task(Future):
    """ A coroutine wrapped in a Future. """

# 이말은 태스크로 실행할 때 기다리지 않고 실행된다는 것입니다.
# 예제로 확인해봅시다.

@asyncio.coroutine
def hello(msg):
    yield from asyncio.sleep(1)
    yield from asyncio.sleep(1)
    yield from asyncio.sleep(1)
    print(msg)

start = time.perf_counter()
loop = asyncio.get_event_loop()
loop.run_until_complete(hello("안녕!"))
end = time.perf_counter()

print(f"elapsed time : {end - start}")
```

yield from coroutine vs yield from task

```
import asyncio

# 이번에는 태스크를 만들어서 실행해 보겠습니다.

import time

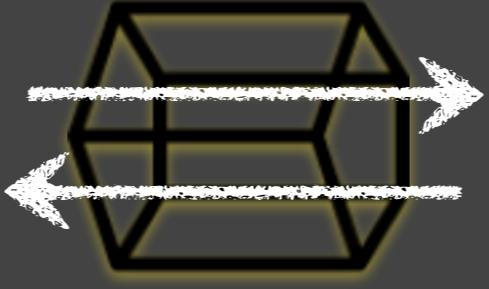

@asyncio.coroutine
def hello_delay(delay, say):
    yield from asyncio.sleep(delay)
    print(say)


def main():
    task1 = asyncio.create_task(hello_delay(1, "hello"))
    task2 = asyncio.create_task(hello_delay(1, "world"))
    task3 = asyncio.create_task(hello_delay(1, "banana"))

    yield from task1
    yield from task2
    yield from task3


start = time.perf_counter()
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
end = time.perf_counter()

print(f"elapsed time : {end - start}")
```



asyncio

- concurrency vs parallelism
- yield from coroutine vs yield from task
- **yield from -> async await**
- spinner 예제
- clock 예제
- 파일 다운로드 받기
- 파일 다운로드 여러개 받기

yield from -> async await

```
import asyncio

# 3.6부터는 사실상 yield from 을 사용하지 않아도 됩니다.
# yield from 은 우리 기억에만 남겨둡시다.
# yield from 으로 했던것들을 asyncio 로 변경해봅시다.
import time

async def hello_delay(delay, say):
    await asyncio.sleep(delay)
    print(say)

async def main():
    task1 = asyncio.create_task(hello_delay(1, "안녕"))
    task2 = asyncio.create_task(hello_delay(1, "async 의 세계로 온것을"))
    task3 = asyncio.create_task(hello_delay(1, "환영해"))

    await task1
    await task2
    await task3

start = time.perf_counter()
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
end = time.perf_counter()

print(f"elapsed time : {end - start}")
```

yield from -> async await

```
import asyncio

# 3.6부터는 사실상 yield from 을 사용하지 않아도 됩니다.
# yield from 은 우리 기억에만 남겨둡시다.
# yield from 으로 했던것들을 asyncio 로 변경해봅시다.
import time

async def hello_delay(delay, say):
    await asyncio.sleep(delay)
    print(say)

async def main():
    task1 = asyncio.create_task(hello_delay(1, "안녕"))
    task2 = asyncio.create_task(hello_delay(1, "async 의 세계로 온것을"))
    task3 = asyncio.create_task(hello_delay(1, "환영해"))

    await task1
    await task2
    await task3

start = time.perf_counter()
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
end = time.perf_counter()

print(f"elapsed time : {end - start}")
```

어?! event_loop?

yield from -> async await

```
import asyncio

# 3.6부터는 사실상 yield from 을 사용하지 않아도 됩니다.
# yield from 은 우리 기억에만 남겨둡시다.
# yield from 으로 했던것들을 asyncio 로 변경해봅시다.
import time

async def hello_delay(delay, say):
    await asyncio.sleep(delay)
    print(say)

async def main():
    task1 = asyncio.create_task(hello_delay(1, "안녕"))
    task2 = asyncio.create_task(hello_delay(1, "async 의 세계로 온것을"))
    task3 = asyncio.create_task(hello_delay(1, "환영해"))

    await task1
    await task2
    await task3

start = time.perf_counter()
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
end = time.perf_counter()

print(f"elapsed time : {end - start}")
```

어?! event_loop?

이벤트 루프는 모든 asyncio 응용 프로그램의 핵심입니다.
이벤트 루프는 비동기 태스크 및 콜백을 실행하고
네트워크 IO 연산을 수행하며 자식 프로세스를 실행합니다.

yield from -> async await

async 가 어떻게 실행되고 결과값을 주는지 한번 볼까요?

```
async def greeting(name):
    return "hello " + name

print(greeting("suengkyoo"))

# output
<coroutine object greeting at 0x106c355c8>
```

yield from -> async await

async 가 어떻게 실행되고 결과값을 주는지 한번 볼까요?

```
async def greeting(name):
    return "hello " + name

print(greeting("suengkyoo"))

# output
<coroutine object greeting at 0x106c355c8>
```

async 가 붙어 있는 함수는 코루틴입니다.
저 return 값은 어떻게 받아올까요?

yield from -> async await

```
# async 가 어떻게 실행되고 결과값을 주는지 한번 볼까요?
```

```
async def greeting(name):  
    return "hello " + name
```

```
coro = greeting("Andy")  
coro.send(None)
```

```
# output  
Traceback (most recent call last):  
  File "/Users/gyus/dev/python/pycon2019/async_example/async4.py", line 9, in <module>  
    coro.send(None)  
StopIteration: hello Andy
```

yield from -> async await

```
# async 가 어떻게 실행되고 결과값을 주는지 한번 볼까요?
```

```
async def greeting(name):  
    return "hello " + name
```

```
coro = greeting("Andy")  
coro.send(None)
```

```
# output  
Traceback (most recent call last):  
  File "/Users/gyus/dev/python/pycon2019/async_example/async4.py", line 9, in <module>  
    coro.send(None)  
StopIteration: hello Andy
```

coroutine에서 결과값을 받아 올때와 마찬가지로
StopIteration의 에러에 값을 담아줍니다.

run이라는 함수를 만들어서 StopIteration의 값을 리턴하도록 해봅시다.

yield from -> async await

```
# StopIteration에서 값을 받아봅시다.
```

```
async def greeting(name):  
    return "hello " + name
```

```
def run(coro):  
    try:  
        coro.send(None)  
    except StopIteration as e:  
        return e.value
```

```
result = run(greeting("파이콘kr 2019"))
```

```
print(result)
```

```
# output
```

```
hello 파이콘kr 2019
```

yield from -> async await

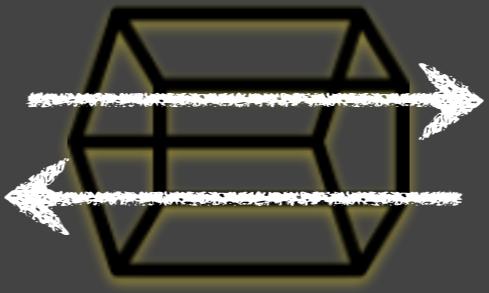
```
# run도 원래 파이썬에 있는걸 가져다 써볼게요.
import asyncio

async def greeting(name):
    return "hello " + name

async def main():
    result = await greeting("pyconkr 2019")
    print(result)

asyncio.run(main())

# output
hello pyconkr 2019
```



asyncio

- concurrency vs parallelism
- yield from coroutine vs yield from task
- yield from -> async await
- **spinner 예제**
- **clock 예제**
- 파일 다운로드 받기
- 파일 다운로드 여러개 받기

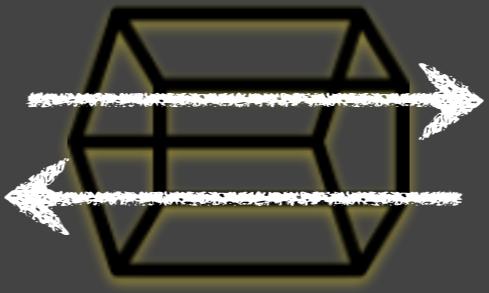
spinner 예제

```
import asyncio
import itertools

"""빵글이 만들기"""
async def spin(msg):
    for char in itertools.cycle("|/-\\"):
        status = char + " " + msg
        print(status, flush=True, end="\r")
        try:
            await asyncio.sleep(0.1)
        except asyncio.CancelledError:
            break
        print(" " * len(status), end="\r")

async def runner():
    spinner = asyncio.create_task(spin("wait for it~"))
    print("spinner ", spinner)
    await asyncio.sleep(3)
    spinner.cancel() # 스레드는 Task 를 중단할 수 없지만, async await 는 가능하다.
    print("Legendary~~~! ")

result = asyncio.run(runner())
```



asyncio

- concurrency vs parallelism
- yield from coroutine vs yield from task
- yield from -> async await
- spinner 예제
- **clock** 예제
- 파일 다운로드 받기
- 파일 다운로드 여러개 받기

clock 예제

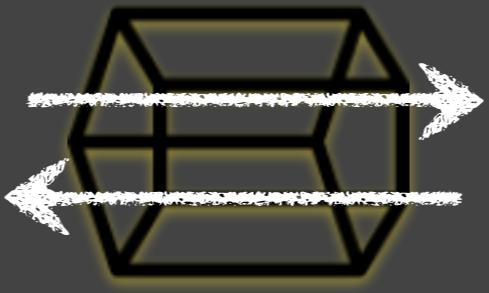
```
import asyncio
import itertools
import time

async def seconds():
    while True:
        for i in range(60):
            print(i)
            await asyncio.sleep(1)

async def minute():
    for i in range(1, 10):
        await asyncio.sleep(60)
        print(i, "min")

async def main():
    await asyncio.gather(seconds(), minute())

asyncio.run(main())
```

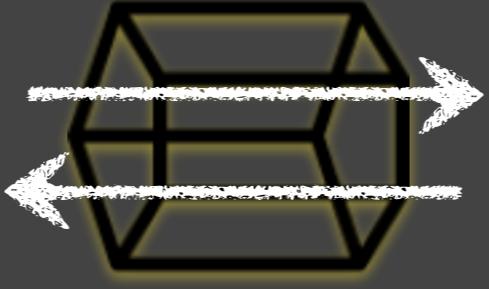


asyncio

- concurrency vs parallelism
- yield from coroutine vs yield from task
- yield from -> async await
- spinner 예제
- clock 예제
- 파일 다운로드 받기**
- 파일 다운로드 여러개 받기**

파일 다운로드 받기

```
# 파일에서 기본으로 제공해주는 asyncio 라이브러리는 http 프로토콜을 지원하지 않습니다.  
# aiohttp라는 서드파티 라이브러리를 사용해야해요~  
# 아래의 명령어로 설치후 진행합시다.  
# pip3 install aiohttp  
# 이번에는 파일을 하나 다운로드 받아보는 것을 해볼거예요.  
# http://pycon.gyus.me/test.txt 파일을 받아봅시다.  
  
import asyncio  
import aiohttp  
  
async def aioget(url):  
    async with aiohttp.ClientSession() as session:  
        async with session.get(url) as res:  
            return await res.text()  
  
BASE_URL = "http://dev.gyus.me/"  
  
async def main():  
    result = await aioget(BASE_URL + "test.txt")  
    print(result)  
  
asyncio.run(main())
```



asyncio

- concurrency vs parallelism
- yield from coroutine vs yield from task
- yield from -> async await
- spinner 예제
- clock 예제
- 파일 다운로드 받기
- 파일 다운로드 여러개 받기**

파일 여러개 다운로드 받기

```
# 이번에는 여러개를 다운받는 것을 해보도록 합시다.  
# tmp 라는 디렉토리를 만들고 그 디렉토리에 파일을 저장해 봅시다.  
# 원래는 웹페이지를 파싱하는 내용이 있었지만, 생략했습니다.  
# beautifulsoup 같은 라이브러리를 참고하세요~  
  
import asyncio  
import aiohttp  
  
async def aioget(url, is_binary=False):  
    async with aiohttp.ClientSession() as session:  
        async with session.get(url) as res:  
            if is_binary:  
                return await res.read()  
            return await res.text("UTF-8")  
  
BASE_URL = "http://dev.gyus.me/"  
  
async def main():  
  
    for i in range(1, 13):  
        url = f"{BASE_URL}{i}.png"  
        print(url)  
        img = await aioget(url, True)  
        with open("./tmp/{i}.png", "wb") as f:  
            f.write(img)  
  
asyncio.run(main())
```

파일 여러개 다운로드 받기

```
# 이번에는 여러개를 다운받는 것을 해보도록 합시다.  
# tmp 라는 디렉토리를 만들고 그 디렉토리에 파일을 저장해 봅시다.  
# 원래는 웹페이지를 파싱하는 내용이 있었지만, 생략했습니다.  
# beautifulsoup 같은 라이브러리를 참고하세요~  
  
import asyncio  
import aiohttp  
  
async def aioget(url, is_binary=False):  
    async with aiohttp.ClientSession() as session:  
        async with session.get(url) as res:  
            if is_binary:  
                return await res.read()  
            return await res.text("UTF-8")  
  
BASE_URL = "http://dev.gyus.me/"  
  
async def main():  
  
    for i in range(1, 13):  
        url = f"{BASE_URL}{i}.png"  
        print(url)  
        img = await aioget(url, True)  
        with open(f"./tmp/{i}.png", "wb") as f:  
            f.write(img)  
  
asyncio.run(main())
```

여러개를 받을 수는 있긴한데...
빠르지는 않을겁니다.
await에서 매번 대기하고 있으니까요
조금 더 빠른 버전을 만들어봅시다.

파일 여러개 다운로드 받기

```
# gather 를 이용해서 여러개의 태스크를 동시에 실행할 수 있습니다.
import asyncio
import aiohttp

async def aioget(url, is_binary=False):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as res:
            if is_binary:
                return await res.read()
            return await res.text("UTF-8")

BASE_URL = "http://dev.gyus.me/"

async def download_one(url, filename):
    print(url, filename)
    img = await aioget(url, True)
    with open(f"./tmp/{filename}", "wb") as f:
        f.write(img)

async def main():
    tasks = []
    for i in range(1, 13):
        filename = f"{i}.png"
        url = f"{BASE_URL}{filename}"
        tasks.append(download_one(url, filename))
    await asyncio.gather(*tasks)

asyncio.run(main())
```

파일 여러개 다운로드 받기

```
# future 의 iterator 를 리턴하는 as_completed 를 사용해도 좋습니다.
import asyncio
import aiohttp

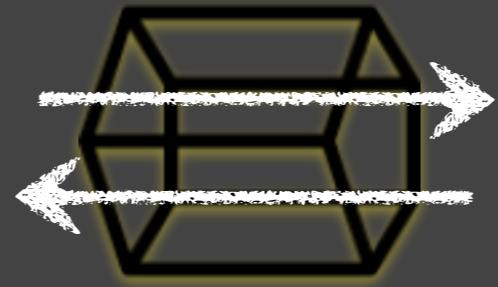
async def aioget(url, is_binary=False):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as res:
            if is_binary:
                return await res.read()
            return await res.text("UTF-8")

BASE_URL = "http://dev.gyus.me/"

async def download_one(url, filename):
    print(url, filename)
    img = await aioget(url, True)
    with open(f"./tmp/{filename}", "wb") as f:
        f.write(img)

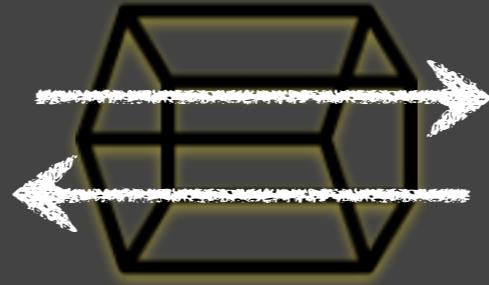
async def main():
    tasks = [download_one(f"{BASE_URL}{i}.png", f"{i}.png") for i in range(1, 13)]
    for future in asyncio.as_completed(tasks):
        await future

asyncio.run(main())
```



더 알아 보기

- future
- threadpool executor, processpool executor
- async iterator, async context manager
- socket 을 사용하는 예제
- asyncio의 다른 api 들
- curio, trio, twisted등의 다른 스케줄러



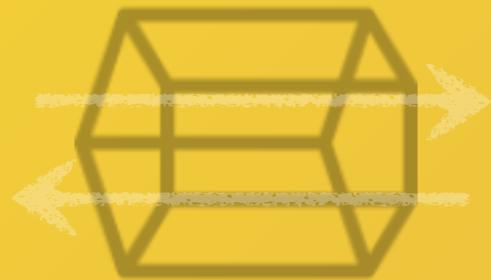
참고자료

- 전문가를 위한 파이썬 14, 17, 18장
- [Python Concurrency From the Ground Up: LIVE! - PyCon 2015](#)
- [Fear and Awaiting in Async](#)
- [Curious Course on Coroutines and Concurrency](#)



@decorator

데코레이터



async

비동기



meta
programming

메타 프로그래밍

"메타클래스"는 사용자의 99%가 걱정하는 것보다
더 깊은 심연의 마법이다. 만약 여러분이 이것이 필요한지
궁금하다면, 여러분에게는 필요가 없는 것이다.

1%의 영역

"메타클래스"는 사용자의 99%가 걱정하는 것보다
더 깊은 심연의 마법이다. 만약 여러분이 이것이 필요한지
궁금하다면, 여러분에게는 필요가 없는 것이다.

- Tim peters -

1%의 영역

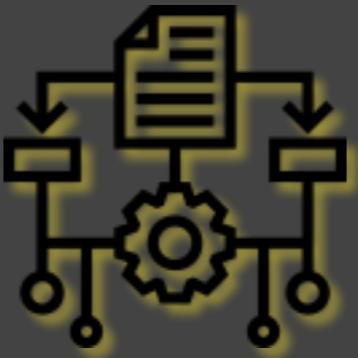
"메타클래스
더 깊은 심연의
궁금하다면"

사람들이 잘 사용하지 않는 것이지
어렵다는 것은 아닙니다

파이썬스럽지 않은 어색함을 조금 참고
그냥 해보면 됩니다.

~~just do it~~

하는 것보다
이것이 필요한지
는 것이다.



meta programming

- 간략하게 얘기하면 코드를 조작하는 코드
- 데커레이터, 메타클래스, 디스크립터 같은 것들이 있다.
- 프레임워크, 라이브러리 제작시 도움이 된다.

메타클래스

- 메타클래스를 알려면 클래스를 자세히 들여다 보아야 합니다.

이름	상속
----	----

```
class Klass(object):  
    pass  
    body
```

이름, 상속, body 를 잘 기억해 둡시다

```
cls = Klass    변수에 할당  
print(cls)
```

```
obj = Klass()    실행  
print(obj)
```

```
<class '__main__.Klass'>  
<__main__.Klass object at 0x1092c5860>
```

메타클래스

- 클래스를 동적으로 만들어 봅시다.

```
def make_class(name):
    if name == "pdf":
        class PdfFile:
            pass
        return PdfFile
    else:
        class TxtFile:
            pass
        return TxtFile

PdfFile = make_class("pdf")
print(PdfFile)
print(PdfFile())

<class '__main__.make_class.<locals>.PdfFile'>
<__main__.make_class.<locals>.PdfFile object at 0x105654240>
```

메타클래스

- 클래스를 동적으로 만들어 봅시다.

```
def make_class(name):
    if name == "pdf":

        class PdfFile:
            pass

        return PdfFile
    else:

        class TxtFile:
            pass

        return TxtFile
```

미리 정의해놓지 않으면 써먹을 수 없으니 동적이라고 말하기는 좀 그렇습니다.

```
PdfFile = make_class("pdf")
print(PdfFile)
print(PdfFile())
```



```
<class '__main__.make_class.<locals>.PdfFile'>
<__main__.make_class.<locals>.PdfFile object at 0x105654240>
```

메타클래스

type(obj) 써보셨나요?

메타클래스

```
# type 함수로 타입검사.  
# instance 를 넣으면 Class 를 알려준다.  
# 아래는 신기하게도 다 type 이다.  
# 즉 int, str, set, tuple 는 type의 인스턴스라는 얘기다.  
# type 은 클래스를 만드는 클래스 : 즉 메타 클래스이다.  
print(type(int))  
print(type(str))  
print(type(list))  
print(type(set))  
print(type(tuple))  
print(type(type)) # type의 type도 type 이다...  
  
# 얘도 type 이다.  
print(type(Klass))  
  
# 실행을 하면 Klass 가 나온다.  
print(type(Klass()))  
  
# 값  
print(type(1))  
print(type("1"))
```

메타클래스

type의 두번째 사용법이 있습니다

type(name, bases, attrs)

name : 클래스명

bases : 클래스의 부모 클래스들의 튜플

attrs : 클래스의 속성 딕셔너리

메타클래스

```
# type으로 클래스를 만들어보자
Water = type("Water", (), {"taste": "무맛", "color": "투명", "state": "liquid"})

print(Water) # class
print(Water()) # object
print(Water.taste)
print(Water.color)

# 부모클래스도 넣어보자
Cola = type("Cola", (Water,), {"taste": "콜라맛", "color": "black", "price": 500})
print(Cola)
print(Cola())
print(Cola.price)

def is_liquid(self):
    return self.state == "liquid"

SparklingWater = type(
    "SparklingWater", (Water,), {"taste": "탄산맛", "is_liquid": is_liquid, "price": 600}
)

print(hasattr(Cola, "state")) # true
print(hasattr(SparklingWater, "state")) # true

spwater = SparklingWater()
print(spwater.is_liquid())
```

메타클래스

```
def is_liquid(self):
    return self.state == "liquid"

SparklingWater = type(
    "SparklingWater", (Water,), {"taste": "탄산맛", "is_liquid": is_liquid, "price": 600}
)

print(hasattr(Cola, "state")) # true
print(hasattr(SparklingWater, "state")) # true

spwater = SparklingWater()
print(spwater.is_liquid())

# 동적으로 함수를 추가 할 수도 있어요.
def discount10(self):
    return self.price * 0.9

SparklingWater.discount = discount10
print(hasattr(SparklingWater, "discount")) # true
```

메타클래스

클래스를 만들어주는 무엇 : ????

메타클래스

클래스를 만들어주는 무엇: 메타클래스

네 바로 `type`의 정체는 메타클래스였습니다
파이썬에서는 이에 대한 편리한 문법을 제공합니다.

메타클래스

```
# meta class 에서는 __init__ 보다는 __new__ 를 사용합니다.  
# 사용법은 아래와 같습니다.  
# __new__ (<클래스자신>, <클래스명>, (클래스의 부모 클래스), {클래스의 어트리뷰트 딕셔너리} )  
# __new__ 가 실행된 다음에 __init__ 가 실행되게 됩니다.  
  
class Meta(type):  
    def __new__(cls, name, bases, attrs):  
        print("__new__ 메서드!")  
        print(cls, name, bases, attrs)  
        return type.__new__(cls, name, bases, attrs)  
  
    def __init__(cls, name, bases, attrs):  
        print("__init__ 메서드")  
        type.__init__(cls, name, bases, attrs)  
  
    print("=====")  
    print("<메타클래스가 초기화 됩니다.>")  
    class MyClass(metaclass=Meta):  
        pass  
    print("=====")  
  
# print 로 찍은 값을 보시면 그저 클래스를 정의만 했는데  
# 메타클래스가 어딘가 생성된것을 볼 수 있습니다.
```

메타클래스

```
# meta class 에서는 __init__ 보다는 __new__ 를 사용합니다.  
# 사용법은 아래와 같습니다.  
# __new__ (<클래스자신>, <클래스명>, (클래스의 부모 클래스), {클래스의 어트리뷰트 딕셔너리} )  
# __new__ 가 실행된 다음에 __init__ 가 실행되게 됩니다.  
  
class Meta(type):  
    def __new__(cls, name, bases, attrs):  
        print("__new__ 메서드!")  
        print(cls, name, bases, attrs)  
        return type.__new__(cls, name, bases, attrs)  
  
    def __init__(cls, name, bases, attrs):  
        print("__init__ 메서드")  
        type.__init__(cls, name, bases, attrs)  
  
    print("=====")  
    print("<메타클래스가 초기화 됩니다.>")  
    class MyClass(metaclass=Meta):  
        pass  
    print("=====")  
  
# print 로 찍은 값을 보시면 그저 클래스를 정의만 했는데  
# 메타클래스가 어딘가 생성된것을 볼 수 있습니다.
```

예제를 하나 더 봅시다.

메타클래스

```
## django 코드의 일부분입니다.  
class ModelBase(type):  
    """  
    Metaclass for all models.  
    """  
    def __new__(cls, name, bases, attrs):  
        super_new = super().__new__  
  
        app_label = None  
  
        if getattr(meta, 'app_label', None) is None:  
            raise RuntimeError(  
                "Model class %s.%s doesn't declare an explicit "  
                "app_label and isn't in an application in "  
                "INSTALLED_APPS." % (module, name)  
    )
```

django의 ModelBase 는 메타클래스입니다
이 코드는 모델에 `app_label` 이 없으면,
`RuntimeError` 를 내도록 하고 있습니다.
간단하게 따라 만들어 봅시다.

메타클래스

```
class ModelMeta(type):
    def __new__(cls, name, bases, attrs):
        print(cls, name, bases, attrs)
        if "app_label" not in attrs:
            raise RuntimeError(f"{name} 클래스에 app_label 을 정의해주세요!")
        return type.__new__(cls, name, bases, attrs)
```

```
class ModelBase(metaclass=ModelMeta):
    app_label = ""
```

```
class GoodModel(ModelBase):
    app_label = "good_app"
```

```
m = GoodModel()
```

```
print(m.app_label)
```

```
# 여기서 에러가 납니다.
```

```
class BadModel(ModelBase):
    pass
```

메타클래스가 `type()` 보다 좋은점은
상속이 된다는 것입니다
코드를 조금 더 깔끔하게 유지 할 수 있습니다

메타클래스

```
class ModelMeta(type):
    def __new__(cls, name, bases, attrs):
        print(cls, name, bases, attrs)
        if "app_label" not in attrs:
            raise RuntimeError(f"{name} 클래스에 app_label 을 정의해주세요!")
        return type.__new__(cls, name, bases, attrs)
```

```
class ModelBase(metaclass=ModelMeta):
    app_label = ""
```

```
class GoodModel(ModelBase):
    app_label = "good_app"
```

```
m = GoodModel()
```

```
print(m.app_label)
```

```
# 여기서 에러가 납니다.
```

```
class BadModel(ModelBase):
    pass
```

일일이 만드는 것은 귀찮으니, 클래스명으로 무조건
추가하는 것을 기본값으로 넣어 줍시다.
이때 `__prepare__(cls, name, bases)`
를 사용해 보겠습니다.

메타클래스

```
class ModelMeta(type):
    def __new__(cls, name, bases, attrs):
        if 'app_label' not in attrs:
            raise RuntimeError(f"{name} 클래스에 app_label 을 정의해주세요!")
        return type.__new__(cls, name, bases, attrs)

    @classmethod
    def __prepare__(cls, name, bases):
        return {'app_label': name}

class ModelBase(metaclass=ModelMeta):
    app_label = ''

class GoodModel(ModelBase):
    app_label = 'good_app'

m = GoodModel()

print(m.app_label)
```

`__prepare__()` 는 `__new__()` 와 다르게
클래스 메서드로 정의해줘야합니다

```
class NotBadModel(ModelBase):
    pass

m2 = NotBadModel()
print(m2.app_label)
```

**클래스의 생성과 상속의 커스터마이징
조금은 해볼 수 있겠죠?**

이제 속성에 대한 것을 알아봅시다

이제 속성에 대한 것을 알아봅시다

저는 이전에는 클래스나 인스턴스의 속성값을
읽고 변경하는 것에 대해 신경써본적이 거의 없었습니다.
같이 알아가 봅시다

vars(obj), dir(obj), __dict__로 클래스 들여다보기

```
class MyClass:  
    cls_attr = "클래스 속성이예요 인스턴스를 만들지 않고 읽을 수 있죠"  
  
    def __init__(self, param):  
        self.param = param  
  
    print(MyClass)  
    print(MyClass.cls_attr)  
  
obj = MyClass("변수1")  
  
print(obj) # 인스턴스의 주소가 기본적으로 나옵니다.  
print(obj.param) # 인스턴스의 변수의 값입니다.  
  
# vars(obj) 혹은 dir(obj) 를 사용하면 해당 오브젝트의 속성들을 들여다 볼 수 있습니다.  
print(vars(MyClass))  
print("=====  
print(vars(obj))  
  
# 변수들은 어디있나요?  
  
print("=====  
print(MyClass.__dict__)  
print(obj.__dict__)
```

vars(obj), dir(obj), __dict__ 로 클래스 들여다보기

```
class MyClass:  
    cls_attr = "클래스 속성이예요 인스턴스를 만들지 않고 읽을 수 있죠"  
  
    def __init__(self, param):  
        self.param = param  
  
    print(MyClass)  
    print(MyClass.cls_attr)  
  
obj = MyClass("변수1")  
  
print(obj) # 인스턴스의 주소가 기본적으로 나옵니다.  
print(obj.param) # 인스턴스의 변수의 값입니다.  
  
# vars(obj) 혹은 dir(obj) 를 사용하면 해당 오브젝트의 속성들을 들여다 볼 수 있습니다.  
print(vars(MyClass))  
print("=====  
print(vars(obj))  
  
# 변수들은 어디있나요?  
print("=====  
print(MyClass.__dict__)  
print(obj.__dict__)
```

속성을 읽는 동작을 커스터마이징
하려면 뭘 하면 좋을까요?

property

```
property (fget, fset, fdel, doc)
```

property

```
class Person:

    name = property() # name 속성은 이제 프로퍼티입니다.
    change_count = 0
    max_change = 2

    def __init__(self, name):
        self._name = name

    @name.getter # 값을 가져오는 동작을 커스터마이징 합니다.
    def name(self):
        print("이름을 가져옵니다...")
        return self._name

    @name.setter # 값을 변경하는 동작을 커스터마이징 합니다.
    def name(self, tobe_name):
        print("이름 변경")
        if self.change_count >= self.max_change:
            raise ValueError("개명은 3번이상할 수 없습니다.")

        self.change_count += 1
        self._name = tobe_name

me = Person("andy")
me.name = '생구이'
print(me.name)

me.name = 'Andy'
print(me.name)

me.name = 'Woodie'
```

property

```
class Person:

    name = property() # name 속성은 이제 프로퍼티입니다.
    change_count = 0
    max_change = 2

    def __init__(self, name):
        self._name = name

    @name.getter # 값을 가져오는 동작을 커스터마이징 합니다.
    def name(self):
        print("이름을 가져옵니다...")
        return self._name

    @name.setter # 값을 변경하는 동작을 커스터마이징 합니다.
    def name(self, tobe_name):
        print("이름 변경")
        if self.change_count >= self.max_change:
            raise ValueError("개명은 3번이상할 수 없습니다.")

        self.change_count += 1
        self._name = tobe_name

me = Person("andy")
me.name = '생구이'
print(me.name)

me.name = 'Andy'
print(me.name)

me.name = 'Woodie'
```

뭔가 자바스러워졌습니다...
파이썬이라면 "원모얼땅" 이 있을
것 같은데요??

Descriptor Protocol

Property 는 사실 Descriptor 로 구현한 것입니다.

descriptor

```
class Descriptor:

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

    def __get__(self, instance, cls):
        print("__get__")
        try:
            return instance.__dict__[self.name]
        except KeyError:
            return None

    def __set__(self, instance, value):
        print("__set__", value)
        instance.__dict__[self.name] = value

    def __delete__(self, instance):
        print("__del__")
        del instance.__dict__[self.name]

# 이렇게 사용합니다.
class User:
    name = Descriptor('name')

u = User()
u.name # name.__get__(u, User)
u.name = "andy" # name.__set__(u, "andy")

del u.name # name.__delete__(name)
```

descriptor

```
class Descriptor:

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

    def __get__(self, instance, cls):
        print("__get__")
        try:
            return instance.__dict__[self.name]
        except KeyError:
            return None

    def __set__(self, instance, value):
        print("__set__", value)
        instance.__dict__[self.name] = value

    def __delete__(self, instance):
        print("__del__")
        del instance.__dict__[self.name]

# 이렇게 사용합니다.

class User:
    name = Descriptor('name')

    u = User()
    u.name # name.__get__(u, User)
    u.name = "andy" # name.__set__(u, "andy")

    del u.name # name.__delete__(name)
```

디스크립터를 여러개 사용하는
예제를 만들어 봅시다.

descriptor

```
# 디스크립터 여러개 사용하기
class SeriesProduct:
    title = Descriptor("title")
    cost_per_page = Descriptor("cost_per_page")
    is_waitfree = Descriptor("is_waitfree")
    author = Descriptor("author")

    sp = SeriesProduct()
    sp.title = "나혼자만 레벨업"
    sp.author = "추공"
    sp.cost_per_page = 100
    sp.is_waitfree = True

    print(sp.__dict__)
```

descriptor

```
# 디스크립터 여러개 사용하기
class SeriesProduct:
    title = Descriptor("title")
    cost_per_page = Descriptor("cost_per_page")
    is_waitfree = Descriptor("is_waitfree")
    author = Descriptor("author")

sp = SeriesProduct()
sp.title = "나훈자만 레벨업"
sp.author = "추공"
sp.cost_per_page = 100
sp.is_waitfree = True

print(sp.__dict__)
```

속성에 할당했지만, Descriptor는 "클래스"입니다
기존 Descriptor 를 상속받아서
validation 체크용 Descriptor를 만들어봅시다.

descriptor

```
# 디스크립터 여러개 사용하기
class SeriesProduct:
    title = Descriptor("title") 읽기전용
    cost_per_page = Descriptor("cost_per_page") 100원 단위로 할당가능한 디스크립터
    is_waitfree = Descriptor("is_waitfree") True / False 만 할당가능하게
    author = Descriptor("author") 읽기전용

sp = SeriesProduct()
sp.title = "나훈자만 레벨업"
sp.author = "추공"
sp.cost_per_page = 100
sp.is_waitfree = True
print(sp.__dict__)
```

속성에 할당했지만, Descriptor는 "클래스"입니다
기존 Descriptor 를 상속받아서
validation 체크용 Descriptor를 만들어봅시다.

descriptor

```
class ReadOnly(Descriptor):
    # set, delete 시 에러나게 하면 됩니다.
    def __set__(self, instance, value):
        raise ValueError("읽기전용입니다.")

    def __delete__(self, instance):
        raise ValueError("읽기전용입니다.")


class Positive(Descriptor):
    def __set__(self, instance, value):
        if isinstance(value, int) and value > 0:
            super().__set__(instance, value)
        else:
            raise ValueError("양수만 세팅가능합니다.")


class HundredWon(Descriptor):
    def __set__(self, instance, value):
        if isinstance(value, int) and value % 100 == 0:
            super().__set__(instance, value)
        else:
            raise ValueError("100원 단위로만 세팅가능합니다.")


class Boolean(Descriptor):
    def __set__(self, instance, value):
        if isinstance(value, bool):
            super().__set__(instance, value)
        else:
            raise ValueError("True | False 만 세팅가능합니다.")
```

descriptor

```
class ReadOnly(Descriptor):    읽기전용
    # set, delete 시 에러나게 하면 됩니다.
    def __set__(self, instance, value):
        raise ValueError("읽기전용입니다.")

    def __delete__(self, instance):
        raise ValueError("읽기전용입니다.")
```

```
class Positive(Descriptor):    양수만
    def __set__(self, instance, value):
        if isinstance(value, int) and value > 0:
            super().__set__(instance, value)
        else:
            raise ValueError("양수만 세팅가능합니다.")
```

```
class HundredWon(Descriptor):    100원단위
    def __set__(self, instance, value):
        if isinstance(value, int) and value % 100 == 0:
            super().__set__(instance, value)
        else:
            raise ValueError("100원 단위로만 세팅가능합니다.")
```

```
class Boolean(Descriptor):    True | False 만
    def __set__(self, instance, value):
        if isinstance(value, bool):
            super().__set__(instance, value)
        else:
            raise ValueError("True | False 만 세팅가능합니다.")
```

descriptor

```
# 시리즈 클래스를 조금 수정해 봅시다.  
class SeriesProduct:  
    title = ReadOnly("title")  
    cost_per_page = HundredWon("cost_per_page")  
    is_waitfree = Boolean("is_waitfree")  
    author = ReadOnly("author")  
  
    def __init__(self, title, author):  
        self.__dict__["title"] = title  
        self.__dict__["author"] = author
```

Readonly 라 값을 바로 할당할 수 없어서 **__dict__** 에 할당하는 꼼수를....썼습니다.

```
sp = SeriesProduct("나혼자만 레벨업", "추공")
```

```
print(sp.__dict__)  
sp.cost_per_page = 100  
sp.is_waitfree = True
```

```
print(sp.__dict__)
```

```
sp.title = "독고"
```

descriptor

```
# 시리즈 클래스를 조금 수정해 봅시다.  
class SeriesProduct:  
    title = ReadOnly("title")  
    cost_per_page = HundredWon("cost_per_page")  
    is_waitfree = Boolean("is_waitfree")  
    author = ReadOnly("author")  
  
    def __init__(self, title, author):  
        self.__dict__["title"] = title  
        self.__dict__["author"] = author  
  
sp = SeriesProduct("나혼자만 레벨업", "추공")  
  
print(sp.__dict__)  
sp.cost_per_page = 100  
sp.is_waitfree = True  
  
print(sp.__dict__)  
sp.title = "독고"
```

한 가지 문제가 있는데 눈치채신분??

descriptor

```
# 시리즈 클래스를 조금 수정해 봅시다.  
class SeriesProduct:  
    title = ReadOnly("title")  
    cost_per_page = HundredWon("cost_per_page")  
    is_waitfree = Boolean("is_waitfree")  
    author = ReadOnly("author")  
  
    def __init__(self, title, author):  
        self.__dict__["title"] = title  
        self.__dict__["author"] = author  
  
sp = SeriesProduct("나혼자만 레벨업", "추공")  
  
print(sp.__dict__)  
sp.cost_per_page = 100  
sp.is_waitfree = True  
  
print(sp.__dict__)  
  
sp.title = "독고"  
  
sp.cost_per_page = -100  
print(sp.__dict__)
```

가격을 마이너스로 설정이 가능합니다..

descriptor

```
# 시리즈 클래스를 조금 수정해 봅시다.  
class SeriesProduct:  
    title = ReadOnly("title")  
    cost_per_page = HundredWon("cost_per_page")  
    is_waitfree = Boolean("is_waitfree")  
    author = ReadOnly("author")  
  
    def __init__(self, title, author):  
        self.__dict__["title"] = title  
        self.__dict__["author"] = author  
  
sp = SeriesProduct("나훈자만 레베인", "독고")  
print(sp.__dict__)  
sp.cost_per_page = 100  
sp.is_waitfree = True  
  
print(sp.__dict__)  
  
sp.title = "독고"  
  
sp.cost_per_page = -100  
  
print(sp.__dict__)
```

두개의 클래스를 상속받는
Descriptor 를 만들어 봅시다.

descriptor

```
# 두개의 클래스를 상속 받는 Descriptor 를 만들어봅시다.
```

```
class PositiveHundred(Positive, HundredWon):  
    pass
```

```
# 시리즈 클래스를 조금 수정해 봅시다.
```

```
class SeriesProduct:  
    title = ReadOnly("title")  
    cost_per_page = PositiveHundred("cost_per_page") # 여기를 변경했어요.  
    is_waitfree = Boolean("is_waitfree")  
    author = ReadOnly("author")  
  
    def __init__(self, title, author):  
        self.__dict__["title"] = title  
        self.__dict__["author"] = author
```

```
sp = SeriesProduct("나홀자만 레벨업", "장경락")
```

```
try:  
    sp.cost_per_page = -100  
except ValueError as e:  
    print(e)
```

```
try:  
    sp.cost_per_page = 101  
except ValueError as e:  
    print(e)
```

어디서 많이 본 모양 아니신가요?

descriptor

```
# 두개의 클래스를 상속 받는 Descriptor 를 만들어봅시다.
```

```
class PositiveHundred(Positive, HundredWon):  
    pass
```

```
# 시리즈 클래스를 조금 수정해 봅시다.
```

```
class SeriesProduct:  
    title = ReadOnly("title")  
    cost_per_page = PositiveHundred("cost_per_page") # 여기를 변경했어요.  
    is_waitfree = Boolean("is_waitfree")  
    author = ReadOnly("author")  
  
    def __init__(self, title, author):  
        self.__dict__["title"] = title  
        self.__dict__["author"] = author
```

```
sp = SeriesProduct("나홀자만 레벨업", "장경락")
```

django의 Model 설정이 디스크립터를 닮았습니다.

```
try:  
    sp.cost_per_page = -100  
except ValueError as e:  
    print(e)
```

```
try:  
    sp.cost_per_page = 101  
except ValueError as e:  
    print(e)
```

descriptor

```
# 두개의 클래스를 상속 받는 Descriptor 를 만들어봅시다.
```

```
class PositiveHundred(Positive, HundredWon):  
    pass
```

```
# 시리즈 클래스를 조금 수정해 봅시다.
```

```
class SeriesProduct:  
    title = ReadOnly("title")  
    cost_per_page = PositiveHundred("cost_per_page") # 여기를 변경했어요.  
    is_waitfree = Boolean("is_waitfree")  
    author = ReadOnly("author")  
  
    def __init__(self, title, author):  
        self.__dict__["title"] = title  
        self.__dict__["author"] = author
```

```
sp = SeriesProduct("나홀자만 레벨업", "장경락")  
  
try:  
    sp.cost_per_page = -100  
except ValueError as e:  
    print(e)
```

```
try:  
    sp.cost_per_page = 101  
except ValueError as e:  
    print(e)
```

하나씩 접근하는 경우 말고,
모든 변수에 접근하는 경우를
커스터마이징 하 수 있을까요?
가능합니다. 다만 조심해서 사용하세요~

`__getattribute__`, `__setattr__`

```
__getattribute__(self, name)
__setattr__(self, name, value)
```

`__getattr__` 이
`__setattr__` 과 짹이 아니라는 건
좀 이상하긴 합니다

__getattribute__, __setattr__

```
import logging
logger = logging.getLogger(__name__)

# 빌링같은 데이터는 민감한 데이터라 오브젝트의 값을 읽을때마다 로그를 남겨야합니다.
# 민감한 클래스의 변수에 접근할 때마다 로그를 남기도록 해봅시다.

class BillingComponent(object):
    def __init__(self, user):
        self.user = user

    def __getattribute__(self, key):
        # 메서드 몸체에 self로 변수에 접근을 하면 안됩니다.
        # 왜냐하면 변수에 접근할 때 __getattribute__를 다시 호출하게되어, 무한 루프에 빠지게 됩니다.
        user = super().__getattribute__('user')
        try:
            if key != 'user':
                print(f"[DEBUG] user <{user}> try to GET key '{key}'")
            return super().__getattribute__(key)
        except AttributeError:
            logger.error(f"<{user}> try to get prohibited key : {key}")
            return None

    def __setattr__(self, key, val):
        if key != 'user':
            user = super().__getattribute__('user')
            print(f"[DEBUG] user <{user}> try to SET '{key}': '{val}'")
        super().__setattr__(key, val)

    def __getattr__(self, key):
        pass # __getattribute__ 와 어떻게 다를까요?

bill = BillingComponent('andy')
print("=>> ", bill.user)
print("=>> ", bill.password)
bill.receipt = '100원'
```

__getattribute__, __setattr__

```
import logging
logger = logging.getLogger(__name__)

# 빌링같은 데이터는 민감한 데이터라 오브젝트의 값을 읽을때마다 로그를 남겨야합니다.
# 민감한 클래스의 변수에 접근할 때마다 로그를 남기도록 해봅시다.

class BillingComponent(object):
    def __init__(self, user):
        self.user = user

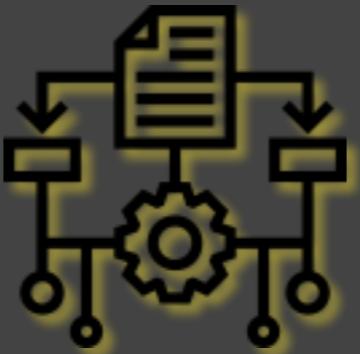
    def __getattribute__(self, key):
        # 메서드 몸체에 self로 변수에 접근을 하면 안됩니다.
        # 왜냐하면 변수에 접근할 때 __getattribute__를 다시 호출하게되어, 무한 루프에 빠지게 됩니다.
        user = super().__getattribute__('user')
        try:
            if key != 'user':
                print(f"[DEBUG] user <{user}> try to GET key '{key}'")
            return super().__getattribute__(key)
        except AttributeError:
            logger.error(f"<{user}> try to get prohibited key : {key}")
            return None

    def __setattr__(self, key, val):
        if key != 'user':
            user = super().__getattribute__('user')
            print(f"[DEBUG] user <{user}> try to SET '{key}': '{val}'")
        super().__setattr__(key, val)

    def __getattr__(self, key):
        pass # __getattribute__ 와 __setattr__가 같은 경우에만 실행된다.

bill = BillingComponent('andy')
print("=>> ", bill.user)
print("=>> ", bill.password)
bill.receipt = '100원'
```

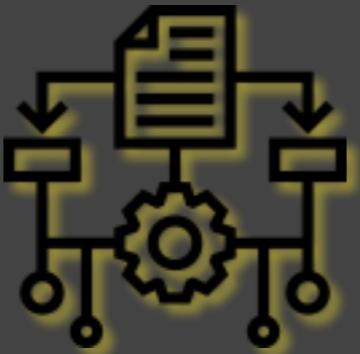
__getattribute__은 없는 속성에 접근할 때 실행됩니다.
유효성 검증을 넣어주거나,
없는 값을 다른곳에서 가져와서
세팅해 줄 수 있습니다.



meta programming

더 알아보기

- AST : Abstract Syntax Trees
- inspect
- import system
- dis : Disassembler for python bytecode



meta programming

참고자료

- 파이썬 언어 레퍼런스 : 3. 데이터 모델
- Python3 Metaprogramming (데이터베이스 비즈니스)
- Python metaclasses by example
- effective python
- python cookbook 3rd

**좋은 파이썬 개발자로
함께 자라갑시다**

감사합니다