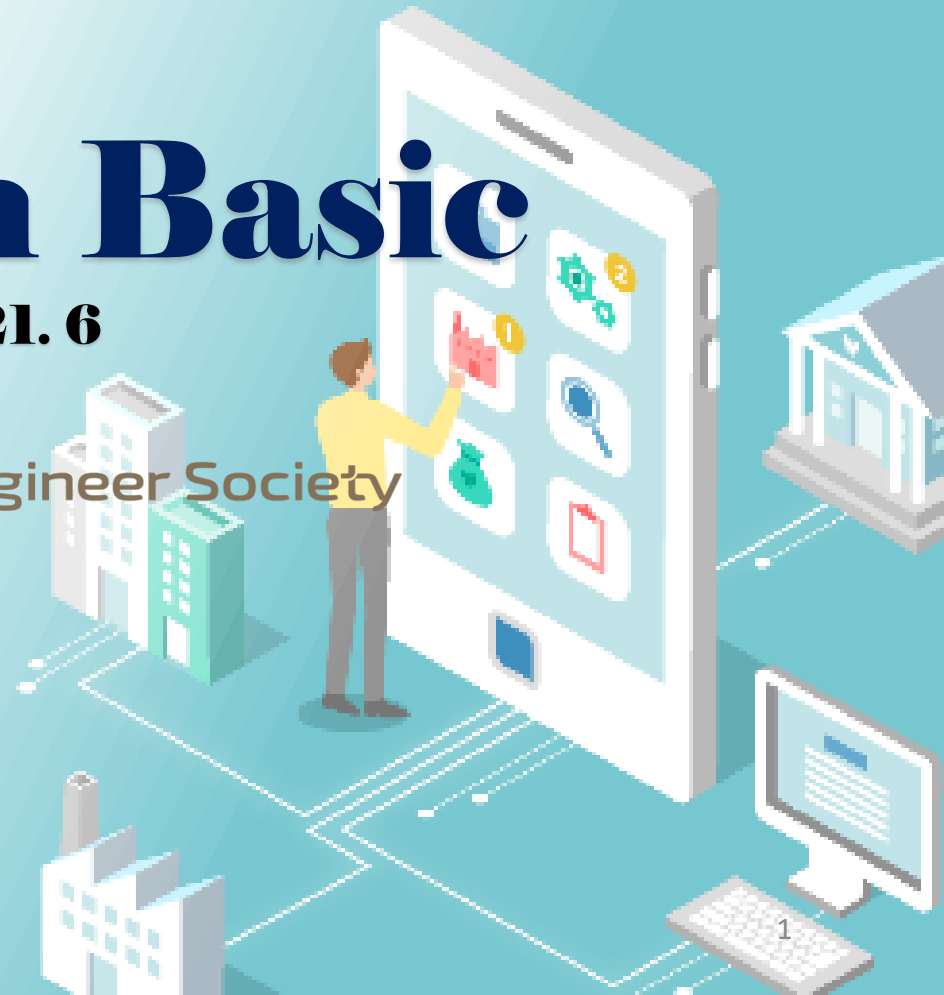


Python Basic

2021. 6



Soft Engineer Society



▶ filter() 함수

- iterable 객체에서 요소를 하나씩 가져와 조건과 비교한 결과를 filter 객체로 반환하는 함수.
- 조건을 함수로 작성하여 전달.

```
filter(함수, iterable 객체)
```

```
test_result = {"홍길동": 90, "전우치": 80, "사오정": 30}
```

```
def judge(name):  
    return test_result[name] > 70
```

```
pass_list = list(filter(judge, test_result))  
print(pass_list)
```

▶ map() 함수

- iterable 객체에서 요소를 하나씩 가져와 일괄 처리한 결과를 map 객체로 반환하는 함수.
- 처리식은 함수로 작성하여 전달.
- iterable 객체를 두 개 이상 전달하여 조합 가능.

```
map(함수, iterable 객체[, ...])
```

```
product_list = ["TV", "냉장고", "세탁기"]  
price_list = [1000, 2000, 3000]
```

```
def discount(product, price):  
    return (product, price // 2)
```

```
sale_list = list(map(discount, product_list, price_list))  
print(sale_list)
```

▶ reduce() 함수

- iterable 객체에서 요소를 하나씩 가져와 단계적으로 처리한 결과를 반환하는 함수.
- 처리식은 함수로 작성하여 전달.
- python 3부터 내장 함수에서 제외.

```
from functools import reduce  
reduce(함수, iterable 객체)
```

```
from functools import reduce  
  
char_list = list("python")  
  
def concat(a, b):  
    return a + b  
  
str = reduce(concat, char_list)  
print(str)
```

▶ 람다식

- 이름 없는 함수를 만드는 식.
- 일회성 함수.

lambda 인수[, 인수2, ...]: 표현식

```
def sum(x, y):  
    return x + y  
sum(10, 20)
```

```
(lambda x, y: x + y)(10, 20)
```

▶ 람다식 예제

```
test_result = {"홍길동": 90, "전우치": 80, "사오정": 30}
```

```
pass_list = list(filter(lambda name: test_result[name] > 70, test_result))  
print(pass_list)
```

```
product_list = ["TV", "냉장고", "세탁기"]  
price_list = [1000, 2000, 3000]
```

```
sale_list = list(map(lambda product, price: (product, price // 2), product_list, price_list))  
print(sale_list)
```

```
from functools import reduce
```

```
char_list = list("python")
```

```
str = reduce(lambda a, b: a + b, char_list)  
print(str)
```

▶ iterator의 원리

- iterable 객체는 반복을 요청하는 구문을 만났을 때 `__iter__()` 함수로 iterator 객체를 생성함.
- 생성된 iterator 객체는 반복이 이루어질 때마다 `__next__()` 함수로 다음 요소를 반환함.
- 더 이상 반환할 요소가 없으면 `StopIteration` 예외를 발생시킴.

```
for num in [1, 2, 3]:  
    print(num)
```

```
iter = [1, 2, 3].__iter__()  
while True:  
    try:  
        num = iter.__next__()  
    except StopIteration:  
        break  
    print(num)
```

▶ 사용자 정의 iterator 예제

```
class Serial:
    def __init__(self, max):
        self.max = max

    def __iter__(self):
        return Serial_iterator(self.max)

class Serial_iterator:
    def __init__(self, max):
        self.max = max
        self.index = -1

    def __next__(self):
        self.index += 1
        if self.index >= self.max:
            raise StopIteration
        return self.index
```

```
for i in Serial(10):
    print(i)
```


▶ generator

- return 대신 yield로 값을 반환하는 함수.
- 함수 내에서 사용하는 변수의 마지막 값과 상태를 저장함.
- 함수 실행 결과 generator 객체가 반환됨.
- generator 객체 내부에서 자동으로 `__iter__()`, `__next__()` 함수를 생성하여 iterator처럼 동작함.

```
def serial_generator(max):  
    for index in range(0, max):  
        yield index  
  
for i in serial_generator(10):  
    print(i)
```

일급 객체 (First-class object)

▶ 일급 객체란?

- 다른 객체들에 일반적으로 적용 가능한 연산을 모두 지원하는 객체.
- 어떤 함수를 다른 값처럼 전달할 수 있다면 해당 함수를 일급 시민 혹은 일급 함수라고 칭함.

```
def add(a, b):  
    return a + b
```

```
plus = add  
print(plus(10, 20))
```

```
def calc(op, a, b):  
    return op(a, b)
```

```
def add(a, b):  
    return a + b
```

```
def multi(a, b):  
    return a * b
```

```
print(calc(add, 10, 20))  
print(calc(multi, 10, 20))
```

▶ 지역 함수 (내부 함수)

- 어떤 함수 내에 존재하는 함수.
- 상위 함수에 존재하는 변수에 접근 가능.

▶ 클로저

- 지역 함수의 특성을 이용한 함수의 형태.
- 어떤 함수 내에 존재하는 변수를 남기기 위해 해당 변수를 반환하는 지역 함수를 선언하여 그 지역 변수 자체를 리턴하도록 만듦.
- 파이썬에선 주로 이미 만들어진 함수를 크게 수정하지 않고 기능을 추가하기 위해 사용.

클로저 (Closure)

▶ 클로저 예제

```
def global_func():  
    var = 10  
    def local_func():  
        return var  
    return local_func  
  
myfunc = global_func()  
print(myfunc)  
print(myfunc())
```

```
def set_tag(tag):  
    tag = tag  
    def set_text(text):  
        return "<%s>%s</%s>" % (tag, text,  
tag)  
    return set_text  
  
temp = set_tag("a")  
result = temp("링크")  
print(result)
```

클로저 (Closure)

▶ nonlocal 키워드

- 내부 함수에서 외부 함수의 지역 변수를 조작할 때 사용.
- nonlocal 키워드가 적용되면 상위 scope 중 가장 가까운 영역의 동명의 지역 변수를 찾아서 인식.

```
def set_num(input):  
    num = input  
    def increase_num():  
        num += 1  
        return num  
    return increase_num  
  
func = set_num(10)  
print(func()) # 익셉션 발생
```

```
def set_num(input):  
    num = input  
    def increase_num():  
        nonlocal num  
        num += 1  
        return num  
    return increase_num  
  
func = set_num(10)  
print(func())
```

▶ 함수 데코레이터

- 일반적으로 이미 정의된 함수의 내용을 수정하지 않으면서 기능을 추가하고자 할 때 wrapper 함수를 사용함.
- 기존 함수명을 그대로 사용하면서 wrapper 함수를 호출할 때 데코레이터를 사용함.

일급 함수를 이용한 예제

```
def wrapper(func):  
    print("Something to say")  
    print("=" * 16)  
    func()  
  
def say():  
    print("hello")  
  
wrapper(say)
```

클로저를 이용한 예제

```
def wrapper():  
    print("Something to say")  
    print("=" * 16)  
    def say():  
        print("hello")  
    return say  
  
inner = wrapper()  
inner()
```

▶ 함수 데코레이터 예제

일급 함수 + 클로저 예제

```
def decorator(func):  
    def wrapper():  
        print("Something to say")  
        print("=" * 16)  
        func()  
    return wrapper
```

```
def say():  
    print("hello")
```

```
inner = decorator(say)  
inner()
```

함수 데코레이터 예제

```
def decorator(func):  
    def wrapper():  
        print("Something to say")  
        print("=" * 16)  
        func()  
    return wrapper
```

```
@decorator  
def say():  
    print("hello")
```

```
say()
```

▶ 함수 데코레이터 예제

```
# 매개변수를 전달하는 경우
def decorator(func):
    def wrapper(*args, **kwargs):
        print("Something to say")
        print("=" * 16)
        func(*args, **kwargs)
    return wrapper
```

```
@decorator
def say():
    print("hello")
```

```
@decorator
def echo(anything):
    print(anything)
```

```
say()
echo("hi")
```


▶ 두 개 이상의 데코레이터 적용 예제

```
def decorator1(func):  
    def wrapper(*args, **kwargs):  
        print("Something to say")  
        print("=" * 16)  
        func(*args, **kwargs)  
    return wrapper
```

```
def decorator2(func):  
    def wrapper(*args, **kwargs):  
        print("Anything to say")  
        print("=" * 16)  
        func(*args, **kwargs)  
    return wrapper
```

```
@decorator2  
@decorator1  
def say():  
    print("hello")  
  
@decorator1  
@decorator2  
def echo(anything):  
    print(anything)  
  
say()  
print()  
echo("hi")
```

▶ 두 개 이상의 데코레이터 적용시 문제

```
def decorator1(func):
    def wrapper(*args, **kwargs):
        print("func.__name__ at deco1:"\
, func.__name__)
        func(*args, **kwargs)
    return wrapper

def decorator2(func):
    def wrapper(*args, **kwargs):
        print("func.__name__ at deco2:"\
, func.__name__)
        func(*args, **kwargs)
    return wrapper
```

```
@decorator2
@decorator1
def say():
    print("hello")

@decorator1
@decorator2
def echo(anything):
    print(anything)

say()
print()
echo("hi")
```

▶ wraps를 이용한 해결

```
from functools import wraps

def decorator1(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("func.__name__ at deco1:"\
, func.__name__)
        func(*args, **kwargs)
    return wrapper

def decorator2(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("func.__name__ at deco2:"\
, func.__name__)
        func(*args, **kwargs)
    return wrapper
```

```
@decorator2
@decorator1
def say():
    print("hello")

@decorator1
@decorator2
def echo(anything):
    print(anything)

say()
print()
echo("hi")
```

▶ 클래스 데코레이터 예제

```
class Decorator:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print("추가할 기능")
        return self.func(*args, **kwargs)

@Decorator
def original_func():
    print("기존 기능")

original_func()
```