

# Ask Django

함수

# Agenda

- Positional/Keyword Arguments
- 가변인자
- 익명함수
- 1급객체와 1급 함수/클래스
- 고차함수

# 함수 (Functions)

# 함수 (Functions)

- 함수는 <작업 지시서> 와 같은 개념
  - 코드 중복을 제거하기 위한 목적
- 함수의 구성
  - 1개의 **함수명** (필수) : 작업의 이름
  - 0개 이상의 **인자값** (옵션) : 작업에 필요한 정보
  - 1개의 **반환값** (옵션) : 작업의 결과를 하나 돌려받습니다.
- 코드의 중복을 제거하기 위해서 가장 필요한 문법
- 빌트인 함수 (Builtin Functions) : print, range 등
- 반환값이 없는 함수호출은 None을 리턴합니다.



# 함수의 유형

- 인자 값, 반환 값 없는 함수

```
>>> def myfn1():  
    result = 1 + 2
```

```
>>> print(myfn1())  
None
```

- 인자 값은 있지만, 반환 값은 없는 함수

```
>>> def myfn2(a, b, c):  
    result = a + b + c
```

```
>>> print(myfn2(1, 2, 3))  
None
```

- 인자 값은 없지만, 반환 값은 있는 함수

```
>>> def myfn3():  
        return 10
```

```
>>> print(myfn3())  
10
```

- 인자 값과 반환 값이 모두 있는 함수

```
>>> def myfn4(a, b, c):  
        result = a + b + c  
        return result
```

```
>>> print(myfn4(1, 2, 3))  
6
```

# Scope (변수의 유효범위)

- 변수가 선언되어, 해당 변수가 영향을 미치는 영역
- 지역 변수 (Local Variable)
  - 함수 안에서 선언되어, 함수 내에서만 활용이 가능한 변수
- 전역 변수 (Global Variable)
  - 함수 밖에서 선언되어, 함수 안에서도 활용이 가능한 변수

## 전역 변수

- 코드의 가독성을 해치므로 **권장하지 않는 방법**
- 주로 상수 (Constant) 목적으로 많이 쓰입니다. 파이썬에서는 따로 상수문법이 따로 없습니다.
  - 상수명을 대문자로 씁니다.
- 변수의 값이 변경되는 경우라면, 그 유효범위를 최소화하여 지역변수를 사용하는 것이 버그 발생확률을 획기적으로 낮출 수 있습니다.



# Arguments (인자)

- 함수가 실행되는데에 필요한 0개 이상의 변수 목록

## Positional Arguments

- 인자의 위치에 기반한 인자

```
def fn_with_positional_arguments(name, age):  
    print("당신의 이름은 {}이며, 나이는 {}입니다.".format(name, age))
```

```
fn_with_positional_arguments('Tom', 10)
```

# Keyword Arguments

- 인자의 이름에 기반한 인자
- 디폴트 인자 문법이 함께 적용 : 함수 호출 시에 해당 인자를 지정하지 않으면, 디폴트 인자값으로 값이 자동지정

```
def fn_with_keyword_arguments(name="", age=0):  
    print("당신의 이름은 {}이며, 나이는 {}입니다.".format(name, age))
```

```
fn_with_keyword_arguments(name='Tom', age=10)  
fn_with_keyword_arguments(age=10, name='Tom')  
fn_with_keyword_arguments(age=10)  
fn_with_keyword_arguments(name='Tom')  
fn_with_keyword_arguments('Tom', 10) # Positional Arguments 로도 적용 가능
```

# Packing

- 인자의 갯수를 제한하지 않고, 다수의 인자를 받을 수 있음
- 다수의 Positional Arguments를 하나의 **tuple**로서 받을 수 있음 (**packing**)

```
def fn2(*colors):    # 0개 이상의 인자를 받을 수 있음.  
    for color in colors:  
        print(color)
```

```
fn2()  
fn2('white')  
fn2('white', 'yellow')  
fn2('white', 'yellow', 'black', 'pink')
```

```
def fn3(color1, color2, *other_colors):    # 2개 이상의 인자를 강요
    print('color1 :', color1)
    print('color2 :', color2)
    for color in other_colors:
        print(color)

fn3('brown', 'green')                      # 최소 2개의 인자 지정이 필요
fn3('brown', 'green', 'white')
fn3('brown', 'green', 'white', 'yellow')
```

# Unpacking

- 인자를 넘길 때 **Sequence** Data Type (리스트/튜플 등) 을 다수의 인자인 것처럼 나눠서 전달 가능 (**unpacking**)

```
colors = ['white', 'yellow', 'black']
```

```
fn2(*colors)
```

```
fn2('brown', 'pink', *colors)
```

```
other_colors = ('violet', 'coral', 'cyan')
```

```
fn2('brown', 'pink', *colors, *other_colors)
```

```
fn3('purple', *('aqua', 'beige', 'black'))
```

```
fn3('purple', *['aqua', 'beige', 'black'])
```

- 가변인자없이 tuple/list 인자 1개로서 전달할 수도 있으나, 함수가 원하는 인자가 명확하게 드러나지 않음.

```
def fn1(colors): # 인자 1개로 받습니다.  
    for color in colors:  
        print(color)
```

```
fn1(['white', 'yellow', 'black'])  
fn1(['white', 'yellow', 'black', 'pink'])  
fn1(['white', 'yellow', 'black', 'pink', 'aqua'])
```

# 가변인자 / Keyword Arguments

- 인자의 갯수를 제한하지 않고, 다수의 인자를 받을 수 있음.
- 다수의 Keyword Arguments를 **dict**으로서 받을 수 있음(**packing**)

```
def fn2(**scores):  
    for key, score in scores.items():  
        print(key, score)
```

```
fn2(apple=10, orange=5)  
fn2(apple=10, orange=5, banana=8)  
fn2(apple=10, orange=5, banana=8, mango=9)
```

```
def fn3(apple=0, **scores):  
    print('apple :', apple)  
    for key, score in scores.items():  
        print(key, score)
```

```
fn3(apple=10, orange=5, banana=8, mango=9)
```

- 함수를 정의할 때, 가변인자를 통해 keyword를 지정하지 않은 인자도 받을 수 있으나, 가급적이면 **인자로 받을 keyword인자를 모두 지정**하는 것이 코드관리에 도움이 됩니다.

```
def fn3(apple=0, orange=0, banana=0):  
    print('apple  :', apple)  
    print('orange :', orange)  
    print('banana :', banana)
```

# 단, 인자를 넘길 때는 가변인자 문법을 활용하면, 유연하게 인자를 지정할 수 있습니다.

```
fn3(apple=10, **{'orange': 10, 'banana': 3})  
fn3(apple=10, orange=8, **{'banana': 3})
```



- 가변인자없이 dict 인자 1개로서 전달할 수도 있으나, 함수가 원하는 인자가 명확하게 드러나지 않음.

```
def fn1(scores):  
    for key, score in scores.items():  
        print(key, score)
```

```
fn1({'apple': 10, 'orange': 5})
```

## 함수 정의 시 가변인자 정의가 유용 할 때

- 차후 클래스 상속에서 부모의 멤버함수를 재정의 (**오버라이딩**) 할 때, 유용하게 쓸 수 있습니다.
- 부모의 멤버함수에서 어떤 인자를 받든지 간에, 나는 (자식 클래스의 멤버함수) 받은 인자 그대로 부모에게 넘겨주겠다.

```
class People(object):
    def say_hello(self, name, age, region1=None, region2=None):
        pass

class Developer(People):
    def say_hello(self, *args, **kwargs):    # 받은 그대로
        super().say_hello(*args, **kwargs)  # 부모에게 패스
        print('부모의 인자의 구성은 나는 모르겠고, 나는 받은 그대로 부모에게 돌려줬다 !!!')
```

인자를 넘길 때, 사전을 unpacking 하여 넘길 수 있습니다.

```
colors = ['white', 'yellow', 'black']  
scores = {'apple': 10, 'orange': 5}  
fn2(*colors, **scores)
```

# Anonymous Function (익명함수)

- 파이썬에서는 lambda 식을 통해 익명함수를 생성

```
>>> lambda x, y: x + y  
<function __main__.<lambda>>
```

```
>>> (lambda x, y: x + y)(1, 2)  
3
```

- return 문은 쓰지 않아도, 마지막 값을 리턴값으로 처리
- 대개 인자로 1줄 함수를 지정할 때, 많이 쓰임
- 일반 함수와 인자처리도 동일하게 처리됩니다. (Positional Arguments, Keyword Arguments)

```
>>> def mysum1(x, y):  
        return x + y
```

```
>>> mysum2 = lambda x, y: x + y    # 위 mysum1 함수와 동일하게 동작
```

```
>>> print(mysum1(1, 2))  
3
```

```
>>> print(mysum2(1, 2))  
3
```

```
>>> mysum3 = lambda *args: sum(args)  
>>> print(mysum3(1, 2, 3, 4, 5, 6, 7))  
28
```

# 1급 객체

다른 객체들에 적용 가능한 연산을 모두 지원하는 객체 (위키피디아)

- 인자로 넘기기, 변수에 대입하기 등
- 종류 : 일급 함수/클래스/컨트롤/타입/데이터타입 등

# 파이썬은 1급 함수/클래스를 지원

- 함수/클래스를 런타임에 생성 가능
- 함수/클래스를 변수에 할당이 가능
- 함수/클래스를 인자나 리턴값으로서 전달 가능

```
>>> mysum1 = lambda x, y: x + y
>>> mysum2 = mysum1
>>> mysum2(10, 20)
30
```

```
>>> def myfn(fn, x, y):
    return fn(x, y)
```

```
>>> myfn(mysum1, 10, 20)
30
```

```
>>> myfn(lambda x, y: x * y, 10, 20)
200
```

# High Order Function (고차함수)

- 다른 함수를 생산/소비하는 함수
- 다른 함수를 인자로 받거나, 그 결과로 함수를 반환하는 함수

```
>>> def base_calculator(base):  
    wrap = lambda x, y: base + x + y  
    return wrap
```

```
>>> calc_10 = base_calculator(10)
```

```
>>> calc_10(1, 2)    # 13  
13
```

```
>>> calc_10(10, 20)  
40
```



A person is sitting on a beach chair, facing away from the camera towards the ocean. The sun is low on the horizon, creating a warm, orange glow. The person is holding a laptop on their lap. The text "Life is short, use Python3/Django." is overlaid on the image in a large, elegant, black serif font.

*Life is short,  
use Python3/Django.*