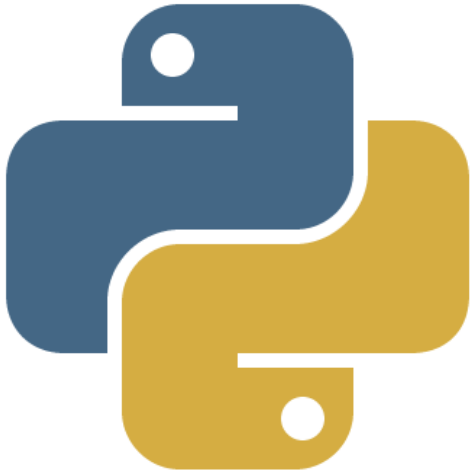
Programming in Python

11 – 함수와 인자



python

2016년 8월, 국민대학교 컴퓨터공학부

함수의 인자 전달 (Argument Passing)

Python 함수 인자 전달의 기본

- 함수의 인자는 자동으로 지역 (local) 변수 이름에 객체를 대입함으로써 이루어진다.
  - 객체에 대한 참조가 전달되는 것이지, 객체의 내용이 복사되지 않음

- 함수 내에서 인자 이름에 대한 대입은 호출한 측에 영향을 미치지 않는다.
  - 인자의 이름은 함수 내의 지역 범위를 가지기 때문

- 함수 내에서 mutable 객체를 변경하는 것은 호출한 측에 영향을 미친다.
  - 함수는 인자로 전달된 mutable 객체를 변경할 수 있음

함수의 인자와 참조의 공유

```
>>> def f(a):
...         a = 99
...

>>> b = 88
>>> f(b)
>>> print(b)
88
```

a 라는 지역 이름 (변수) 에
새로운 값이 대입되면
a 는 인자로 전달된 88 이라는 객체에 대한
참조를 잃어버리고
새로운 객체 99 를 참조하게 됨

```
>>> def changer(a, b):
...     a = 2
...     b[0] = 'spam'
...

>>> X = 1
>>> L = [1, 2]
>>> changer(X, L)
>>> X, L
(1, ['spam', 2])
```

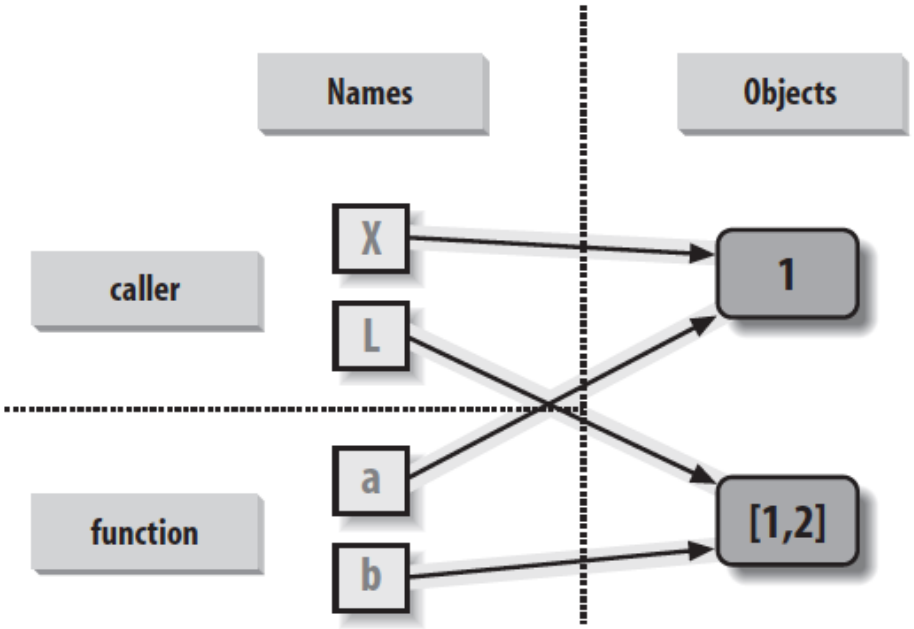
b 는 지역 이름 (변수) 이지만
여전히 L 과 같은 객체 (리스트) 를 참조하고 있으며
따라서 b[0] 에 대한 대입문은
L 에 영향을 미치게 됨

(이 대입문은 b 를 변경하는 것이 아님에 유의!)

인자 전달에 의한 참조 공유

```
>>> def changer(a, b):
...     a = 2
...     b[0] = 'spam'
...

>>> X = 1
>>> L = [1, 2]
>>> changer(X, L)
>>> X, L
(1, ['spam', 2])
```




```
>>> X = 1
>>> a = X
>>> a = 2
>>> print(X)
1
```

```
>>> L = [1, 2]
>>> b = L
>>> b[0] = 'spam'
>>> print(L)
['spam', 2]
```

인자 전달에 의한 변경을 피하려면

- 객체를 전달하지 않고, 객체에 대한 복사본인 새 객체를 전달한다.

```
L = [1, 2]
changer(X, L[:])
```

- 함수 내에서 복사본을 만들어서 이용한다.

```
def changer(a, b):
        b = b[:]
        a = 2
        b[0] = 'spam'
```

- Immutable 객체를 만들어 인자로 전달한다.

```
L = [1, 2]
changer(X, tuple(L))
```

인자 매칭 모드

Python 함수 인자 매칭 방식

- Positionals
  - 인자의 순서에 따라서 매칭 (자동)
- Keywords
  - 인자의 이름을 호출문에 명시함으로써 매칭
- Defaults
  - 주어지지 않는 인자의 디폴트 값을 지정
- Varargs collecting
  - 가변 개수의 인자를 호출되는 함수 쪽에서 수집
- Varargs unpacking
  - 함수 인자를 호출하는 쪽에서 확장

키워드와 디폴트

```
>>> def f(a, b, c):
...     print(a, b, c)
...

>>> f(1, 2, 3)
(1, 2, 3)

>>> f(c=3, b=2, a=1)
(1, 2, 3)

>>> f(1, c=3, b=2)
(1, 2, 3)
```

```
>>> def f(a, b=2, c=3):
...     print(a, b, c)
...

>>> f(1)
(1, 2, 3)

>>> f(a=1)
(1, 2, 3)

>>> f(1, 4)
(1, 4, 3)

>>> f(1, 4, 5)
(1, 4, 5)

>>> f(1, c=6)
(1, 2, 6)
```

```
>>> def func(spam, eggs, toast=0, ham=0):
...     print((spam, eggs, toast, ham))
...

>>> func(1, 2)
(1, 2, 0, 0)

>>> func(1, ham=1, eggs=0)
(1, 0, 0, 1)

>>> func(spam=1, eggs=0)
(1, 0, 0, 0)

>>> func(toast=1, eggs=2, spam=3)
(3, 2, 1, 0)

>>> func(1, 2, 3, 4)
(1, 2, 3, 4)
```

가변 개수 인자를 모으는 두 가지 방법

```
>>> def f(*args):
...     print(args)
...

>>> f()
()

>>> f(1)
(1,)

>>> f(1, 2, 3, 4)
(1, 2, 3, 4)
```

```
>>> def f(**args):
...     print(args)
...

>>> f()
{}

>>> f(a=1, b=2)
{'a': 1, 'b': 2}
```

```
>>> def f(a, *pargs, **kargs):
...     print(a, pargs, kargs)
...

>>> f(1, 2, 3, x=1, y=2)
(1, (2, 3), {'y': 2, 'x': 1})

>>> f(1, 2, x=1, y=2, 3, 4)
SyntaxError: non-keyword arg after keyword arg
```

함수 호출에 있어서 인자를 확장하는 두 가지 방법

```
>>> def func(a, b, c, d):
...     print(a, b, c, d)
...

>>> args = (1, 2)
>>> args += (3, 4)

>>> func(*args)
(1, 2, 3, 4)
```

```
>>> def func(a, b, c, d):
...     print(a, b, c, d)
...

>>> args = {'a': 1, 'b': 2, 'c': 3}
>>> args['d'] = 4

>>> func(**args)
(1, 2, 3, 4)
```

```
>>> func(*(1, 2), **{'d': 4, 'c': 3})
(1, 2, 3, 4)
>>> func(1, *(2, 3), **{'d': 4})
(1, 2, 3, 4)
>>> func(1, c=3, *(2,), **{'d': 4})
(1, 2, 3, 4)
>>> func(1, *(2, 3), d=4)
(1, 2, 3, 4)
>>> func(1, *(2,), c=3, **{'d': 4})
(1, 2, 3, 4)
```

인자 전달 방식의 응용

조건에 따라 서로 다른 함수를
서로 다른 인자로 호출하는 경우

```
if <test>:
        action, args = func1, (1,)
else:
        action, args = func2, (1, 2, 3)
...

action(*args)
```

개발 단계에서 함수의 입출력을 테스트하거나
실행 시간을 측정하는 등, 간접적으로 호출하는 경우

```
def tracer(func, *pargs, **kargs):
        print('calling: ', func.__name__)
        return func(*pargs, **kargs)

def func(a, b, c, d):
        return a + b + c + d

print(tracer(func, 1, 2, c=3, d=4)

calling: func
10
```

실용적 예제 (1)

- 임의의 개수의 임의의 Python 객체가

  인자로 전달될 때, 이 중 minimum 을 찾으시오.

  - 대소관계 연산자는 수치형이 아니라도 됨
  - 따라서, 객체 타입에 따라 별도 함수 필요하지 않음

오른쪽 세 함수 중
어느 것의 성능이 가장 좋을까?

```
def  min1(*args):
        res = args[0]
        for arg in args[1:]:
                if arg < res:
                        res = arg
        return res

def min2(first, *rest):
        for arg in rest:
                if arg < first:
                        first = arg
        return first

def min3(*args):
        tmp = list(args)
        tmp.sort()
        return tmp[0]
```

실용적 예제 (2)

- 임의의 개수의 순차형 Python 객체가 (리스트, 문자열, 순서쌍)

  인자로 전달될 때, 이들의 교집합과 합집합을 구하시오.

  - 인자의 타입이 서로 달라도 동작할 수 있음

```
>>> from xxx import intersect, union
>>> s1, s2, s3 = "SPAM", "SCAM", "SLAM"

>>> intersect(s1, s2), union(s1, s2)
>>> intersect([1, 2, 3], (1, 4))
>>> intersect(s1, s2, s3)
>>> union(s1, s2, s3)
```

```
def intersect(*args):
    res = []
    for x in args[0]:
        for other in args[1:]:
            if x not in other: break
        else:
            res.append(x)
    return res

def union(*args):
    res = []
    for seq in args:
        for x in seq:
            if not x in res:
                res.append(x)
    return res
```

익명 함수: lambda

- lambda 는 문장이 아니라 표현식
  - def 에 의한 함수가 나타날 수 없는 곳에도 적용할 수 있음
  - 예: 리스트의 원소, 함수의 인자 등
- lambda 의 몸체는 문장의 블록이 아니라 하나의 표현식
  - if, while, for 등의 구문 구조를 이용할 수 없음
  - 그러나, Python 의 특징적 표현식을 잘 이용한다면 활용 가치가 높음

```
>>> def func(x, y, z): return x + y + z
...
>>> func(2, 3, 4)
9
```

```
>>> f = lambda x, y, z: x + y + z
>>> f(2, 3, 4)
9
```

lambda 의 응용

```
>>> a1 = {'name': 'John', 'score': 87}
>>> a2 = {'name': 'Peter', 'score': 63}
>>> a3 = {'name': 'Mary', 'score': 92}
>>> L = [a1, a2, a3]

>>> x1 = sorted(L, key=lambda x: x['name'])
>>> x1
[{'score': 87, 'name': 'John'}, {'score': 92, 'name': 'Mary'}, {'score': 63, 'name': 'Peter'}]

>>> x2 = sorted(L, key=lambda x: x['score'], reverse=True)
>>> x2
[{'score': 92, 'name': 'Mary'}, {'score': 87, 'name': 'John'}, {'score': 63, 'name': 'Peter'}]
```

이름 순서로 정렬

점수 순서로 정렬

순차에 대하여 함수 적용: map

```
>>> counters = [1, 2, 3, 4]
>>>
>>> updated = []
>>> for x in counters:
...     updated.append(x + 10)
...
>>> updated
[11, 12, 13, 14]
```

```
>>> def inc(x): return x + 10
...
>>> list(map(inc, counters))
[11, 12, 13, 14]
```

```
>>> [x + 10 for x in counters]
[11, 12, 13, 14]
```

```
>>> pow(3, 4)
81
>>> list(map(pow, [1, 2, 3], [2, 3, 4]))
[1, 8, 81]
```

Quiz

- 다음 각각의 결과는? 그리고 그 이유는?

```
>>> def func(a, b=4, c=5):
...     print(a, b, c)
...
>>> func(1, 2)
```

```
>>> def func(a, b, c=5):
...     print(a, b, c)
...
>>> func(1, c=3, b=2)
```

```
>>> def func(a, *pargs):
...     print(a, pargs)
...
>>> func(1, 2, 3)
```

```
>>> def func(a, **kargs):
...     print(a, kargs)
...
>>> func(a=1, c=3, b=2)
```

Quiz - 정답

- 다음 각각의 결과는? 그리고 그 이유는?

```
>>> def func(a, b=4, c=5):
...     print(a, b, c)
...
>>> func(1, 2)
1, 2, 5
```

```
>>> def func(a, b, c=5):
...     print(a, b, c)
...
>>> func(1, c=3, b=2)
1, 2, 3
```

```
>>> def func(a, *pargs):
...     print(a, pargs)
...
>>> func(1, 2, 3)
1, (2, 3)
```

```
>>> def func(a, **kargs):
...     print(a, kargs)
...
>>> func(a=1, c=3, b=2)
1, {'c': 3, 'b': 2}
```

Quiz

```
L = [{'name': 'John', 'score': 87},
     {'name': 'Peter', 'score': 63},
     {'name': 'Mary', 'score': 92},
]
```

일 때,

점수가 높은 학생부터 이름만 추려서 리스트를 만드는 Python 코드를
lambda 표현식을 이용하여 한 줄로 작성하시오.

즉, 예상 출력 결과는:
['Mary', 'John', 'Peter']

Quiz - 정답

```
L = [{'name': 'John', 'score': 87},
     {'name': 'Peter', 'score': 63},
     {'name': 'Mary', 'score': 92},
]
```

일 때,

점수가 높은 학생부터 이름만 추려서 리스트를 만드는 Python 코드를
lambda 표현식을 이용하여 한 줄로 작성하시오.

```
>>> [x['name'] for x in sorted(L, key=lambda z: z['score'], reverse=True)]
['Mary', 'John', 'Peter']
```

Q & A