11.함수

11.1 함수의 정의

함수는 문들을 묶은 단위이다. 묶은 함수는 더 묶을 수 있다. 프로그램은 계층적으로 묶인 부품들을 만들고, 그것을 이용하여 더 고수준의 부품을 만드는 것과 같다.

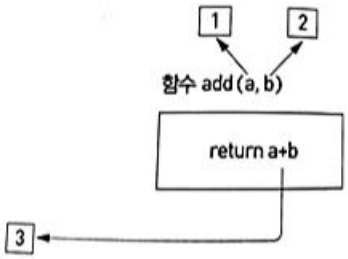
함수는 반복적인 실행이 가능하며, 주위의 상황에 특별히 얽매이지 않는 코드를 만들어 낼 수 있다.
함수는 프로그램을 논리적으로 이해하는 데 도움을 준다.
함수는 단지 반복 실행의 가능성 때문에 정의하는 것이 아니다. 코드의 일정 부분이 별도의 논리적 개념으로 분리하는 것이 가능 할 때 함수 로 분리한다.
함수는 코드를 재사용하게 해주고 프로그램을 논리적으로 구성하게 해준다.

▶ 함수의 정의

```
def 함수이름(인수들):
        <문들>
        return <값>
```

함수를 정의하는 키워드는 def이고 여기에 함수 이름과 괄호 안에 인수들을 적는다.
함수 선언부의 마지막은 항상 콜론(:)으로 끝나야 한다.
함수의 몸체는 (2줄 이상일 때) 그다음 줄에서 시작해야 하며 들여 쓰기를 해야 한다.
파이썬은 어떠한 형식의 데이터도 전달할 수 있으므로 인수의 자료형은 명시하지 않는다.
return 키워드는 계산할 결과 값을 함수를 호출한 곳으로 돌려준다.

```
>>> def add(a, b):       # 함수 정의
...     return a + b     # 값 돌려 주기
...
>>> add                  # 함수 객체 확인
<function add at 0x027DC6A8>
>>> add(1, 2)           # a에 1, b에 2가 전달된다.
3
```



함수는 다른 함수를 호출 할 수 있다.

```
>>> def addabs(a, b):
...     c = add(a, b)
...     return abs(c)
...
>>> addabs(-5, -7)
12
```

함수는 계층 구조로 설계하는데, 입출력이나 기초적인 기능을 하는 함수들 위에 좀 더 고급 기능을 하는 함수를 작성한다.

▶ 자료형의 동적인 결정

파이썬에서는 모든 객체는 동적으로 자료형이 결정되므로, 어떤 연산을 수행할 때 해당 객체에 맞는 연산을 자동으로 호출한다.

예를 들면, +연산은 객체가 숫자인 경우에는 수치 덧셈, 문자열인 경우에는 문자열 연결하기, 리스트인 경우에는 리스트 연결하기 등으로 객체의 + 연산을 정의한 함수를 호출하므로 객체에 맞는 연산을 적용하게 된다.

함수 add는 +연산을 가지는 모든 객체에 적용된다.

```
>>> def add(a, b):       # 함수 정의
...     return a + b     # 값 돌려 주기
...
>>> c  = add(1, 3.4)
>>> d = add('dynamic', 'typing')
>>> e = add(['list'], ['add', 'list'])
>>> print(c, d, e)
4.4 dynamictyping ['list', 'add', 'list']
```

연산자 중복에서 설명하지만, + 연산을 수행하면 각 객체의 __add__ 메소드를 호출한다.

__add__메서드는 객체의 종류마다 자신의 기능을 수행하도록 정의되어야 한다.

따라서 여러분이 정의한 + 연산을 수행하게도 할 수 있다.

```
>>> class MyClass:
...     def __add__(self, b):
...             print('add %s is called' % b)
...
>>> c = MyClass()
>>> c + 1
add 1 is called
>>> c + 'abc'
add abc is called
>>>
```

▶ return 문

인수 없이 return문만을 사용하면 함수를 호출한 측에 아무 값도 전달하지 않는다.

인수 없이 반환을 하지만, 실제로는 None 객체를 전달한다. None객체란 파이썬 내장 객체로서 아무 값도 없음을 표현하는 객체이다.

```
>>> def noting():
...     return
...
>>> def nothing():
...     return
```

```
...
>>> nothing()
>>> print(nothing())
None
```

return 문에 여러 개의 값을 사용할 경우, 이들은 튜플로 구성하여 전달한다.

```
>>> def swap(a, b):
...     return b, a
...
>>> a = 10; b = 20
>>> swap(a, b)
(20, 10)
>>>
```

11.2 함수의 호출

함수로 인수를 전달하는 방법으로는 값에 의한 호출(Call by Value)과 참조에 의한 호출(Call by Reference)이 일반적이다.

값에 의한 호출은 계산한 결과 값이 함수의 인수로 전달되는 것이다. 함수 내에서 인수값이 변경되어도 호출하는 측에 아무런 영향을 미치지 못한다.

이에 반해서 참조에 의한 호출은 호출하는 측의 변수 참조 주소가 호출을 받는 함수의 인수로 잔달된다.

따라서 함수내에서 해당 변수가 변경되면 호출하는 측에도 영향을 미친다.

파이썬에서는 설명한 두 가지 중 어느것도 해당하지 않는 독특한 방법으로 인수를 전달한다.

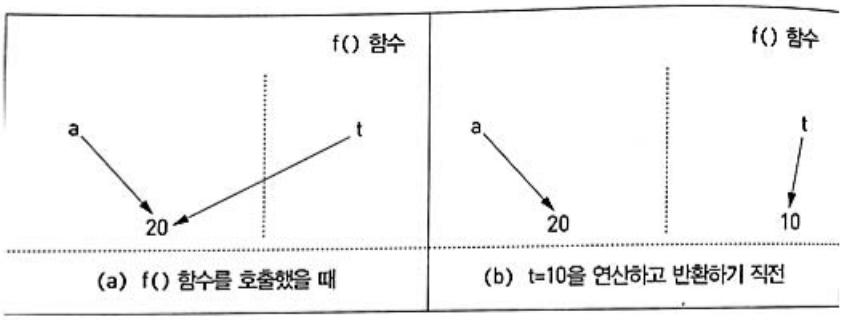
이를 객체 참조에 의한 호출(Call by Object Reference) 혹은 공유에 의한 호출(Call by Sharing)이라고 부른다.

```
>>> def f(t):
...     t = 10
...
>>>
>>> a = 20
>>> f(a)
>>> print(a)
20
```

20이란 객체의 참조를 a가 가지고 있고 이 참조가 함수 f를 통해 t로 잔달되었다.

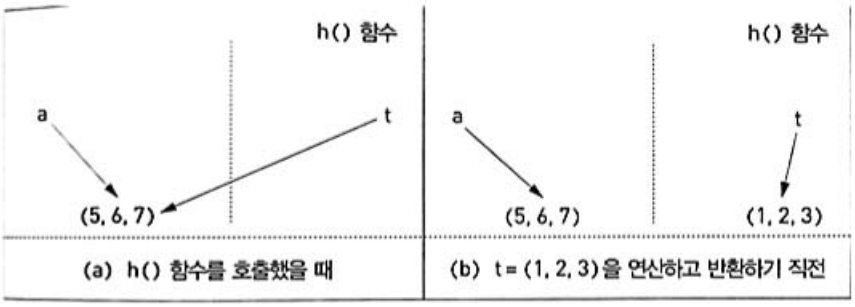
따라서 a와 t는 동일한 참조를 가지고 있다. 이 부분은 참조에 의한 호출(Call by Reference)와 유사하다.

그러나 참조에 의한 호출과는 다르다. a의 값에는 변화가 없고, 단지 10이라는 수치가 생성되고, 이 참조가 이름 t에 치환된다.



이번에는 다른 종류의 변경 불가능한 객체인 튜플을 인수로 넘겨보자.

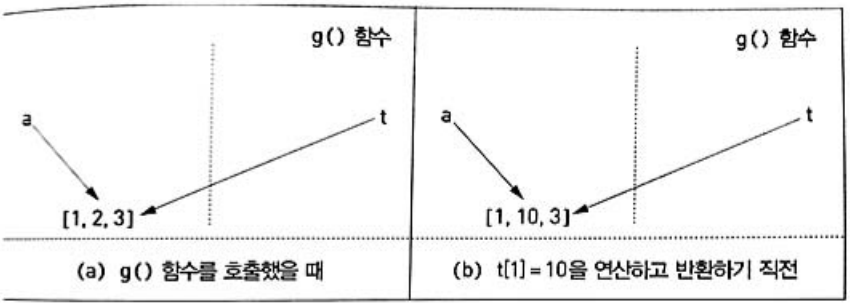
```
>>> def h(t):
...     t = (1, 2, 3)
...
>>> a = (5, 6, 7)
>>> h(a)
>>> a
(5, 6, 7)
>>>
```



함수를 호출할 때 t가 객체 a의 참조를 받지만 함수 내부에서의 치환에 의해 t는 a와는 다른 객체를 참조한다. 결국 a의 값은 변화가 없다.

함수의 인수로 리스트와 같이 변경 가능한 객체를 넘겨주는 경우는 내부 값을 직접 변경할 수 있다.

```
>>> def g(t):
...     t[1] = 10
...
>>> a = [1, 2, 3]
>>> g(a)
>>> print(a)
[1, 10, 3]
>>>
```



함수 g의 인수 t는 리스트 a의 참조를 전달받고 리스트의 두 번째 값 2를 10으로 변경해서 리스트 자체가 변경된다.
a나 t의 참조가 변경되지 않았다. 출력 결과는 [1, 10, 3]이다. 그러나 인수로 전달받은 객체의 참조를 참조하지 않고 다른 객체 값을 치환하는 경우는 앞의 경우와 같이 변경된 내용이 함수를 호출한 측에 반영되지 않는다.

```
>>> def gg(t):
...     t = [1, 2, 3]
...
>>> a = [5, 6, 7]
>>> gg(a)
>>> a
[5, 6, 7]                    ----->   변경되지 않았다.
>>>
```

결론적으로 모든 인수는 인수 자체가(함수 f, h, g와 같이) 다른 객체로 치환될 때, 함수를 호출한 측에 아무런 영향을 미치지 못한다.
변경 가능한 인수는 참조를 이용하여 내부 객체를 변경할 때 변경이 호출한 측에 반영된다. 이러한 파이썬의 독특한 호출 방식을 객체 참조에 의한 호출이나 공유에 의한 호출이라고 한다.

11.3 유효 범위

▶ 규칙

유효 범위 규칙이란 변수가 유효하게 사용되는 문맥(Context)범위를 정하는 규칙이다. 즉, 변수가 특정 범위에서 유효한지를 결정한다.

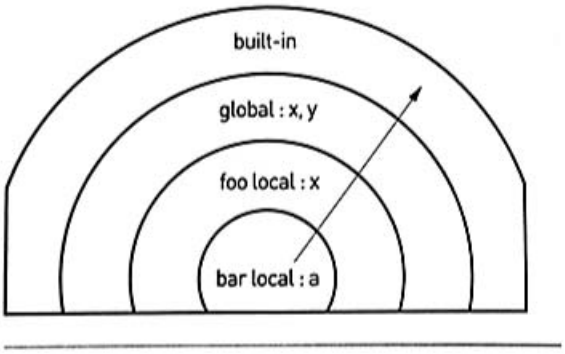
변수는 다양한 이름공간에 저장되어 있는데 파이썬에서는 이름 공간을 찾는 규칙을 LEGB규칙이라고 한다.

- L local 함수 내에 정의된 지역 변수이다.
- E Enclosing Function Local, 함수를 내포하는 또 다른 함수 영역이다.
- G Global, 함수 영역에 포함되지 않은 모듈 영역이다.
- B Built-in, 내장 영역이다.

변수가 저장되는 이름 공간은 변수가 어디에서 정의(혹은 치환)되었는지에 따라서 결정된다. 변수가 함수내에서 정의되면 함수의 지역변수(Local)가 된다. 변수가 함수 외부에서 정의되면 해당 모듈의 전역(Global)변수가 된다.

```
>>> x = 10                # G에 해당한다.
>>> y = 11                # G에 해당한다.
>>> def foo():
...     x = 20           # foo 함수의 L에, bar함수의 E에 해당한다.
...     def bar():
...             a = 30  # L에 해당한다.
...             print(a, x, y)  # 각 변수는 L, E, G에 해당한다.
...     bar()                   # 30 20 11
...     x = 40
...     bar()                   # 30 40 11
...
>>> foo()
30 20 11
30 40 11
```

파이썬에서는 함수 안에 또 다른 함수를 정의하는 것이 가능하다.
예이서 처음 x, y변수응 전역 영역(Global)에 저장되고, foo()함수 안의 x는 foo()함수의 지역(Local)변수 이다.
bar()함수의 이름 공간은 foo()함수의 이름 공간 안에 놓이게 된다.
변수의 이름은 항상 안쪽에서부터 바깥쪽으로 찾아 나간다. 따라서 bar()함수 내에서 x, y변수는 foo()함수의 x와 전역 영역의 y를 참조하게 된다.



유효 범위 규칙

동일한 이름이 있으면 안쪽에 있는 이름 공간의 이름이 먼저 사용되는 것이 원칙이다. abs()라는 내장 함수가 있다.

```
>>> abs
<built-in function abs>
```

모듈 수준에서 abs라는 변수를 새로 정의하면 내장 함수 abs()는 어떻게 될까?
당연히 abs()함수를 사용할 수 없게 된다. 하지만 이 함수는 없어진 것이 아니라 단지 전역 영역에 감추어져 보이지 않을 뿐이다.

```
>>> abs = 10
>>> abs(-5)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
>>> del abs
>>> abs(-5)
5
>>>
```

내장변수 __builtins__을 이용하여 내장 함수를 알 수 있다.

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError',
'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError',
'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError',
'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',
'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError',
'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError',
'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration',
'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
'TimeoutError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', '_', '__build_class__', '__debug__',
'__doc__', '__import__', '__loader__', '__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii',
'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright',
'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format',
'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance',
'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object',
'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr',
'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

▶ global 문

함수 내부에서 값을 치환해서 사용하는 변수를 잔역변수로 사용하려면 어떻게 해야 할까?
global 선언자를 사용하여 변수가 전역 변수임을 선언해야 한다.

```
def f(a):                    # a는 지역변수
        global h
        h = a +  10          # h는 전역 변수
```

전역 변수의 이름을 지으면서 흔히 다음과 같은 실수를 범할 수 있다.

```
>>> g = 10
>>> def f():
...     a = g           # ① g는 지역변수? 전역변수?
...     g = 20          # ② g는 지역변수? 전역변수?
...     return a
...
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
UnboundLocalError: local variable 'g' referenced before assignment
```

②를 살펴보면 함수 내에서 정의된 g는 지역변수이다. 따라서 한 함수 f에 있는 ①의 g는 지역변수이어야 한다. 그런데 ①에서는 지역변수 g의 값이 정해지기도 전에 사용하려고 한 것 이므로 UnboundLocalError에러가 발생한다.
g를 전역변수로 사용하려면 전역 변수로 g를 선언하는 것이다,

```
>>> g = 10
>>> def f():
...     global g        # 전역 변수
...     a = g           # 전역 변수
...     g = 20          # 전역 변수
...     return a
...
>>> f()
10
```

▶ nonlocal 문
전역 영역이 아닌 중첩된 함수의 변수를 사용하려고 할 경우에 변수를 nonlocal문으로 선언할 수 있다.
global 선언지는 전역 영역의 변수에 접근하는 반면, nonlocal 선언자는 가장 가까운 이름 공간에서부터 변수를 찾는다.

```
>>> def outer():
...     x = 1
...     def inner():
...             nonlocal x      # 함수 outer의 x를 사용하게 된다.
...             x = 2           # 함수 inner의 지역 변수가 아니다.
...             print("inner : ", x)
...     inner()
...     print("outer : ", x)
...
>>> outer()
```

```
inner :  2
outer :  2
```

11.4 함수의 인수

▶ 인수의 기본값

인수에서 기본값이란 함수를 호출할 때 인수를 넘겨주지 않아도 인수가 자신의 기본 값을 취하도록 하는 기능이다.
기본값을 지정하면 꼭 필요한 인수만 넘겨주면 되므로 함수 호출이 편리해 진다.
많은 경우에 인수의 값이 고정되어 있거나 기본값이 있는 경우에, 인수에서 기본값을 사용하면 도움이 된다.

예제) incr은 두 개의 인수를 받는다, 두 번째 인수의 기본값이 1로 주어져 있다. 만일 인수를 한 개만 넘기면 두 번째 인수인 step은 값으로 취한다.

```
>>> def incr(a, step = 1):
...     return a + step
...
>>> b = 1
>>> b = incr(b)                 # 1이 증가한다.
>>> b
2
>>> b = incr(b, 10)             # 10이 증가한다.
>>> b
12
```

▶ 키워드 인수

함수의 호출에서 키워드 인수란 인수 이름으로 값을 전달하는 방식이다.

```
>>> def area(height, width):
...     return height * width
...
>>> area(width = 20, height = 10)       # 순서가 아니라 이름으로 값을 전달한다.
200
```

일반적으로 함수를 호출할 때 키워드 인수의 위치는 위치 인수(보통의 인수) 이후이다.

```
>>> def area(height, width):
...     return height - width
...
>>> area(20, width = 5)
15
```

키워드 인수 이후에 순서에 의해서 인수 일치를 시킬 수는 없다.

```
>>> area(width = 5, 20)
  File "<stdin>", line 1
```

```
SyntaxError: positional argument follows keyword argument
>>> area(height = 5, 20)
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
>>>
```

▶ 가변 인수 리스트

고정되지 않은 수의 인수를 함수에 전달하는 방법이 있다.

함수를 정의할 때 인수 목록에 반드시 넘겨야 하는 고정인수를 우선 나열하고, 나머지를 듀플 형식으로 한꺼번에 받는다.

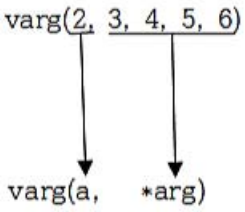
```
>>> def varg(a, *arg):
...     print(a, arg)
...
```

함수 varg를 호출할 때 넘겨지는 첫 인수는 가인수 a가 받으며 나머지는 모두 튜플 형식으로 arg가 받는다.

가변인수는 *var형식으로 인수 목록 마지막에 하나만 나타날 수 있다.

함수 varg를 호출하는 데 예이다.

```
>>> varg(1)
1 ()
>>> varg(2, 3)
2 (3,)
>>> varg(2, 3, 4, 5, 6)
2 (3, 4, 5, 6)
>>>
```



인수가 한 개(1)인 경우 첫 인수 a가 1을 받으며, arg는 빈 튜플을 가진다. 두 번째 경우 첫 인수 2는 a로 3은 튜플 형식으로 arg로 전달한다. 세 번째 경우는 첫 인수 2만 a로 전달되며 나머지(3, 4, 5, 6)는 모두 arg로 전달한다, 이것을 이용하면 파이썬으로 C언어의 printf함수를 그대로 흉내 낼 수 있다.

```
>>> def printf(format, *args):
...     print(format % args)
...
>>> printf("I've spent %d days and %d night to do this", 6, 5)
I've spent 6 days and 5 night to do this
>>>
```

▶ 정의되지 않은 키워드 인수 처리하기

키워드 인수를 이용해서 함수를 호출할 때, 만일 미리 정의되어 있지 않은 키워드 인수를 받으려면 함수를 정의할 때 마지막에 **kw 형식으로 기술한다, 전달받는 형식은 사전이다.

즉, 키는 키워드(변수명)가 되고, 값은 키워드 인수로 전달되는 값이 된다.

```
>>> def f(width, height, **kw):
...     print(width, height)
```

```
...         print(kw)
...
>>> f(width = 10, height = 5, depth = 10, dimension = 3)
10 5
{'depth': 10, 'dimension': 3}
>>>
```

예에서 함수의 가인수로 정의된 width, height 이외의 키워드는 사전 kw에 전달되었다.
이와 같은 사전 키워드 인수는 함수의 가인수 목록의 제일 마지막에 나와야 한다,

```
>>> def f(width, height, **kw):
...         print(width, height)
...         print(kw)
...
>>> f(width = 10, height = 5, depth = 10, dimension = 3)
10 5
{'depth': 10, 'dimension': 3}
>>>
```

▶ 튜플 인수와 사전 인수로 함수를 호출하기

만일 함수 호출에 사용하는 인수들이 튜플에 있으면 *를 이용하여 함수를 호출할 수 있다,

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

만일 함수 호출에 사용하는 인수들이 사전에 있다면 **를 이용하녀 함수를 호출할 수 있다.

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

튜플 인수와 사전 인수를 함께 사용할 수도 있다.

```
>>> args = (1, 2)
>>> dargs = {'c':3}
>>> h(*args, **dargs)
1 2 3
```

11.5 함수 안의 함수

▶ 일급 함수

파이썬의 함수는 모두 일급 함수이다.

일급 함수란 함수를 다른 함수에 인수로 전달할 수 있고, 함수의 반환 값으로 전달할 수 있고, 변수나 자료 구조에 저장할 수 있는 함수를 의미한다.

```
>>> def add(a, b):
...     return a + b
...
>>> addition = add                          # 함수로 저장
>>> addition(3, 4)
7
>>> def  f(g, a, b):                        # 인수로 저장
...     return g(a, b)
...
>>> f(add, 2, 3)
5
>>> def decorate(type = 'italic'):
...     def italic(s):
...             return '<i>' + s + '</i>'
...     def bold(s):
...             return '<b>' + s + '</b>'
...     if type == 'italic':
...             return italic                # 함수로 반환
...     else:
...             return bold                  # 함수로 반환
...
>>> dec = decorate()
>>> dec('hello')
'<i>hello</i>'
```

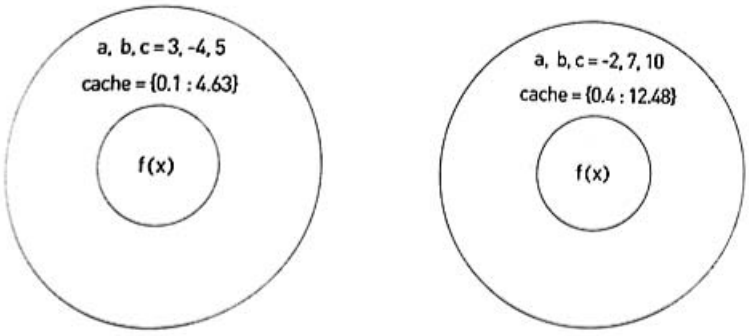
▶ 함수 클로저

함수 클로저(Function Closure)란 함수가 참조할 수 있는 비지역 변수(Non-local Variable)나 자유 변수(Free Variable)를 저장한 심볼 테이블 혹은 참조 환경이 함수와 함께 제공되는 것이다.

```
>>> def quadratic(a, b, c):
...     cache = {}
...     def f(x):
...             if x in cache:
...                     return cache[x]
...             y = a * x * x + b * x + c
...             cache[x] = y
...             return y
```

```
...     return f
...
>>> f1 = quadratic(3, -4, 5)
>>> f1(0.1)
4.63
>>> f2 = quadratic(-2, 7, 10)
>>> f2(0.4)
12.48
```

함수 quadratic의 호출에 의해 반환되는 함수 객체 f는 내부에 지역 변수 x, y를 포함하며 정적인 비지역 변수 cache, a, b, c를 갖는다.



함수 클로저는 함수마다 독립적인 이름 공간을 제공하는 장점이 있다. 이름 이용하면 마치 인스턴스 객체처럼 별도로 동작하는 함수를 정의할 수 있다.

예제) 카운터 함수의 예이다, 두 개의 카운터 함수 c1과 c2는 서로의 간섭 없이 독립적으로 동작한다.

```
>>> def makeCounter():
...     count = 0
...     def counter():
...             nonlocal count
...             count += 1
...             return count
...     return counter
...
>>> c1 = makeCounter()
>>> c2 = makeCounter()
>>> c1()
1
>>> c1()
2
>>> c2()
1
>>>
```

함수 클로저는 함수 객체의 __closure__ 속성으로 확인할 수 있다.

```
>>> c1.__closure__
(<cell at 0x057B1C30: int object at 0x68E5D440>,)
>>> c1.__closure__[0]
<cell at 0x057B1C30: int object at 0x68E5D440>
>>> c1.__closure__[0].cell_contents
2
>>> c1()
3
>>> c1()
4
>>> c1.__closure__[0].cell_contents
4
```

함수 클로저는 클래스의 장식자(Decorator)유용하게 사용하게 된다.

▶ partial()함수

functools모듈에서 제공하는 partial()함수는 함수 클로저를 반환하는 함수이다.
이 함수는 기존 함수를 사용하여 일부 인수가 미리 정해진 새로운 함수를 반환한다.

```
>>> from functools import partial
>>> bin2int = partial(int, base = 2)
>>> bin2int('10010')
18
>>> def quadratic(x, a, b, c):
...     return a * x * x + b * x + c
...
>>> f1 = partial(quadratic, a = 3, b = -4, c = 5)
>>> f1(0.1)
4.63
>>>
```

11.6 한 줄짜리 함수 : 람다 함수

▶ 람다 함수의 정의

람다(Lambda)함수는 이름이 없는 한 줄짜리 함수 이다.
lambda <인수들> : <반환할 식>

예제) 받는 인수가 없고 언제나 1을 반환하는 람다 함수

```
>>> lambda:1                          # 1은 항상 반환할 값을 정의한 것이다.
<function <lambda> at 0x057B38E8>
```

람다 함수는 값을 반환하기 위하여 return 문을 사용하지 않는다. 람다 함수의 몸체(Body)는 문이 아닌 하나의 식이다. 람다 함수는 함수 참조를 반환한다.

새 이름(예, f)으로 람다 함수 객체를 받을 수 있다.
여기서 이름 f는 람다 함수를 참조를 가지고 있으므로 함수 호출에 직접 사용할 수 있다.

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

인수를 가지는 람다 함수를 만들었다,
"x, y"은 함수 인자이고 "x + y"는 반환할 값이다.
두 개의 인수(x, y)를 받으며, 두 값의 합(x + y)을 반환한다,
여기서 return문을 사용하지 않아도 하나의 식으로 이루어지는 결과 값을 반환한다,

람다 함수도 기본값을 가지는 인수와 가변 인수를 지정할 수 있다.

```
>>> incr = lambda x, inc = 1: x + inc
>>> incr(10)                              # inc 기본 인수 값으로 1을 사용한다.
11
>>> incr(10, 5)
15
>>>
>>> vargs = lambda x, *args: args
>>> vargs(1, 2, 3, 4, 5)
(2, 3, 4, 5)
```

정의되지 않은 키워드 인수도 처리할 수 있다.

```
>>> kwords = lambda x, *args, **kw: kw
>>> kwords(1, 2, a = 4, b = 6)
{'a': 4, 'b': 6}
```

▶ 람다 함수를 사용하는 예
람다 함수는 주로 함수를 인수로 넘겨주어야 하는 경우에 편리하다. 일반 함수로 구현한 다음의 예에서 함수 g는 func를 -10 ~ 9범위의 인수로 계산한다.

```
>>> def f1(x):
...     return x * x + 3 * x - 10
...
>>> def f2(x):
...     return x * x * x
...
>>> def g(func):
...     return [ func(x) for x in range(-10, 10)]
```

```
...
>>> g(f1)
[60, 44, 30, 18, 8, 0, -6, -10, -12, -12, -10, -6, 0, 8, 18, 30, 44, 60, 78, 98]
>>> g(f2)
[-1000, -729, -512, -343, -216, -125, -64, -27, -8, -1, 0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
>>>
```

같은 내용을 람다 함수로 구현하면 다음과 같다.

```
>>> def g(func):
...     return [func(x) for x in range(-10, 10)]
...
>>> g(lambda x:x * + 3 * x -10)
[290, 233, 182, 137, 98, 65, 38, 17, 2, -7, -10, -7, 2, 17, 38, 65, 98, 137, 182, 233]
>>> g(lambda x:x * x * x)
[-1000, -729, -512, -343, -216, -125, -64, -27, -8, -1, 0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
>>>
```

일반 함수로 구현한다면 함수를 정의하고 나서 함수를 인수로 전달해야 하지만, 람다 함수는 정의와 동시에 함수 객체로 사용할 수 있다.
이와 같은 차이는 문(Statement)과 식(Expression)에서 온다. 문은 반환값이 없으며, 식의 일부분으로 사용할 수 없다. def 키워드는 문으로 함수를 정의한다, 람다 함수는 식이다, 식은 결과 값이 존재하며 다른 식의 일부로 사용할 수 있다. 따라서 정의와 함께 함수 인수로 전달하는 것이 가능하다.

구분	def로 정의하는 함수	람다 함수
문 / 식	문(Statement)	식(Expression)
함수의 이름	def 다음에 지정한 이름으로 만든 함수 객체를 치환한다.	함수 객체만을 생성한다.
몸체	한 개 이상의 문을 포함한다.	하나의 식만 온다.
반환	return 문에 의해 명시적으로 반환 값을 지정한다.	식의 결과 값을 반환한다.
내부 변수 선언	지역영역에 변수를 만들고 사용하는 것이 가능하다.	지역영역에 변수를 만드는 것이 가능하지 않다.

11.7 함수적 프로그래밍

함수적 프로그래밍(Functional Programming)이란 함수를 사용하여 문제를 해결하는 프로그래밍 방식을 의미한다. 기본적으로 함수는 입력을 받고 출력을 만들어 내는 단위이며, 동일한 입력에 대해서 다른 출력 결과를 만들어 내는 내부 상태를 가지고 있지 않다. 대표적인 언어로는 Standard ML과 OCaml, Haskell 등이 있다.

파이썬은 멀티 패러다임을 추구하는 언어이다. 객체지향 언어이긴 하지만 절차적 언어로 사용하는 것도 가능할 뿐 아니라 함수적 프로그래밍도 가능하도록 설계되었다. 함수적 프로그래밍에서 입력은 일련의 함수들을 계속해서 통과한다. 각 함수는 입력을 받고 출력을 만들어 낸다. 함수적 프로그래밍에서는 내부 상태를 갖는 함수를 지양한다. 같은 입력에 대해 다른 출력을 낼수 있기 때문이다. 또한, 값의 반환 이외에 다른 외부 변수를 변경하는 것도 금한다. 이러한 부작용 없는 함수를 순수 함수(Pure Function)라고 부른다.

순수 함수는 출력 값을 입력에 의존해서만 결정한다. 이러한 점은 객체지향 프로그래밍과는 정반대이다. 객체들은 내부 상태를 가지고 있고 메서드의 호출은 객체의 상태를 반영한 출력을 만들어 낸다. 함수적 프로그래밍은 상태변화를 가능한 한 최소화하고 함수들을 통과해서 만들어지는 데이터로만 작업을 한다. 파이썬에서는 객체들을 입력과 출력으로 하는 순수 함수를 정의함으로써 두 가지 접근 방법을 결합할 수 있다.

반복자(Iterator)는 함수적 프로그래밍에서 중요한 역할을 수행한다. 반복자는 next() 함수를 호출할 때 데이터를 순차적으로 한 번에 하나씩 넘겨주는 자료형이다. 파이썬에서 함수적 프로그래밍에 사용하는 순수 함수는 출력으로 반복자를 반환한다. 이 반복자는 게으른 계산(Lazy Evaluation)을 하기 때문에 출력 값을 필요로 하는 시점에서 값을 계산한다. 따라서 필요한 만큼의 연산을 수행하는 것이 가능하다.

- map() 내장 함수

map() 내장 함수는 입력 집합(X)과 사상 함수(f)가 주어져 있을 때, Y = f(X)를 구한다. 이 함수는 두 개 이상의 인수를 받는다. 첫 인수는 함수(P)이며, 두 번째부터는 입력 집합(X)인 시퀀스 자료형(문자열, 리스트, 튜플 등)이어야 한다. 첫 번째 인수인 함수는 입력 집합 수만큼의 인수를 받는다. 예를 들어, 다음과 같은 코드일 수 있다.

```
>>> def f(X):
...     return x * x
...
>>> X = [1, 2, 3, 4, 5]
>>> map(f, X)           # 반복자 map 객체를 반환한다.
<map object at 0x000001C26ADCC898>
>>> list(X)
[1, 2, 3, 4, 5]
```

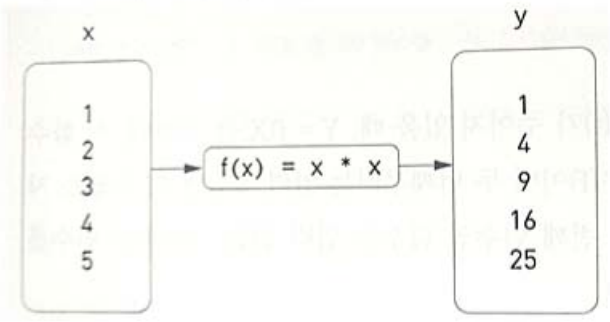
앞의 예에서 map()함수의 첫 번째 인수는 함수(f)이고, 두 번째 인수는 리스트이다. 두 번째 인수의 모든 항목은 첫인수인 함수 F에 적용되고 결과로 반복자인 map 객체를 반환한다.

== 반복자의 게으른 계산

map() 함수의 결과로 반환되는 map 객체는 반복자이다. 반복자는 기본적으로 게으른 계산(Lazy Evaluation)을 수행한다. 즉, 값이 필요한 시점에 실제 계산이 이르어진다는 의미이다. 다음 예를 보자,

```
>>> def f(x):
...     print('calculating..f', x) )
...     return x * x
...
>>> X = [1, 2, 3, 4, 5]
>>> m = map(f, X)                # map 객체를 반환한다.
>>> next(m)                      # 반복자는 next() 함수를 통하여 값을 읽어 낸다.
calculating..f 1                 # 읽어 내는 시점에서 함수가 호출되는 것을 확인한다.
1
>>> next(m)                      # 게으른 계산을 다시 한번 확인한다.
calculating..f 2
4
```

그림. map() 함수의 동작



map()함수는 훨씬 간결하면서도 이해하기 쉬운 코드를 만들어 준다. 개개의 요소를 다루지 않지만, Y=map(F, X)는 집합 X에 함수 f를 적용한 결과 Y를 구한다는 의미가 명확하다. 실행 효율 면에서도 일반 for 문을 사용하는 것보다 앞선다.

이번에는 람다 함수를 사용하여 같은 결과를 얻어 보자.

```
>>> X = [1, 2, 3, 4, 5]
>>> Y = map(lambda a:a * a, X)
>>> Y
<map object at 0x000001C26ADCC908>
>>> list(Y)
[1, 4, 9, 16, 25]
```

앞의 예와 차이점은 첫 번째 인수가 일반 함수가 아닌 람다 함수로 주어졌다는 것이다. 함수를 재사용하지 않을 경우에는 이렇게 간단하게 람다 함수를 이용하는 것이 간편하다.
다른예로 X =ranger(10)의 모든 값 x에 대해서 사상 함수 f=x*x+4*x+5의 결과를 계산해 보자.

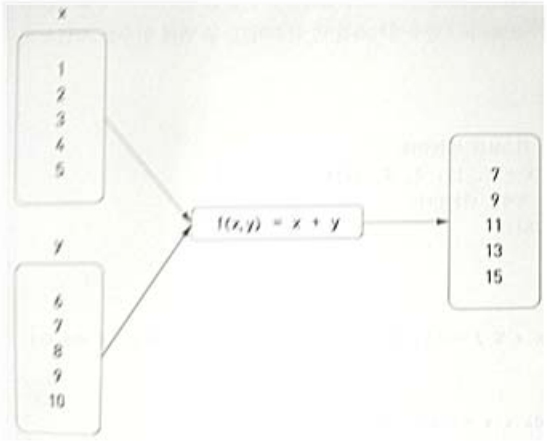
```
>>> X = range(10)
>>> Y = map(lambda x: x * x + 4 * x + 5, X)
```

```
>>> list(Y)
[5, 10, 17, 26, 37, 50, 65, 82, 101, 122]
```

map()함수는 입력을 두 개 이상도 받는다. 이때는 함수도 입력 집합수에 맞추어서 인수를 받아야 한다.

```
>>> X = [1, 2, 3, 4, 5]
>>> Y = [6, 7, 8, 9, 10]
>>> Z = map(lambda x, y:x + y, X, Y)
>>> list(Z)
[7, 9, 11, 13, 15]
```

그림 두 개의 입력에 대한 map() 함수의 동작



만일 map() 함수에 넘겨주는 함수가 앞의 경우와 같이 이미 파이썬에서 정의한 연산일 경우에는 operator 모듈을 이용할 수 있다. 예를 들어, 앞서와 동일한 코드를 다음과 같이 작성할수 있다.

```
>>> X = [1, 2, 3, 4, 5]
>>> Y = [6, 7, 8, 9 ,10]
>>> Z = map(operator.add, X, Y)
>>> list(Z)
[7, 9, 11, 13, 15]
```

operator 모듈에는 다양한 연산 함수가 정의되어 있으므로 자세한 내용은 라이브러리 레퍼런스를 참고하기 바란다.

- filter() 내장함수

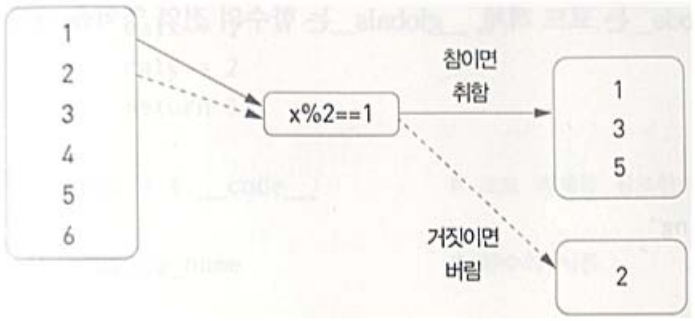
filter() 내장 함수는 주어진 시퀀스형 데이터 중에서 필터링하여 참인 요소만 모아 출력한다.
두 개의 인수를 가지며 첫 인수는(map() 함수에서와같이) 함수이고, 두 번째 인수는 시퀀스 자료형이다.

```
>>> #2 보다 큰 값들로 구성된 리스트를 반환한다.
...
```

```
>>> f = filter(lambda x:x > 2, [1, 2, 3, 34])
>>> f  # 반복자인 filter 객체를 반환한다.
<filter object at 0x000002550F3A3AC8>
>>> list(f)
[3, 34]
>>>
>>>
>>> list(filter(lambda x:x % 2 == 1, [1, 2, 3, 4, 5, 6]))    # 홀수만 반환한다.
[1, 3, 5]
>>>
>>> ' '.join(filter(lambda x:x < 'a', 'abcABCdefDEF')) )))))
'A B C D E F'
```

시퀀스형 데이터의 각 값은 람다 함수에 하나씩 전달되고 결과가 참인 경우만 출력 시퀀스 자료형에 포함된다.

그림 11-8 filter()함수의 동작



조건식이 복잡하면 별도의 함수를 만들어야 할 것이다. 하지만, 이러한 전통적인 방식의 코딩보다는 filter() 함수를 사용함으로써 얻게되는 간결함과 높은 이해도는 코딩하는데 많은 이익을 준다.
다른 유용한 예는 다음과 같이 리스트에서 별 의미없는 값을 삭제하는 것이다.

```
>>> L = ['high', 'level', '', 'built-in', '', 'function']
>>> list(filter(None, L))
['high', 'level', 'built-in', 'function']
```

filter()함수의 첫 번째 인수로 None 객체를 사용하면 입력 값을 진릿값을 판별하는데 그대로 사용할수 있다.

11.8 함수 객체의 속성

함수 객체는 여러 가지 속성을 갖는다. _doc_는 문서 문자열, _name_은 함수의 이름, _defaults_는 기본 인수 값들, _code_는 코드 객체, _globals_는 함수의 전역 영역을 나타내는 사전을 가리킨다.

```
>>> def f(a, b, c = 1):
...     'func attribute testing'
...     localx = 1
...     localy = 2
...     return 1
...
>>> f.__doc__   # 문서 문자열
'func attribute testing'
>>> f.__name__  # 함수의 이름
'f'
>>> f.__defaults__      # 기본 인수 값들
(1,)
>>> f.__code__          # 함수의 코드 객체
<code object f at 0x000002550F2D51E0, file "<stdin>", line 1>
>>> f.__globals__       # 전역 영역
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class
'_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__builtins__': <module
'builtins' (built-in)>, 'operator': <module 'operator' from 'C:₩₩Program
Files₩₩Python36₩₩lib₩₩operator.py'>, 'X': [1, 2, 3, 4, 5], 'Y': [6, 7, 8, 9, 10], 'Z': <map object at
0x000002550F3A3A58>, 'f': <function f at 0x000002550F3A8BF8>, 'L': ['high', 'level', '', 'built-in',
'', 'function']}
```

이 중에서 속성 _code_에 대해서 조금 더 살펴보자. _code_는 함수의 코드 객체이다. 코드객체는 의사 컴파일된 실행 가능한 파이썬 코드이다. 코드 객체는 compile() 내장 함수에 의해 반환되고, 함수 객체의 _code_ 속성으로 참조된다.

코드 객체는 코드에 관한 정보만을 가지고 있는 반면에 함수 객체는 함수를 수애하기 위한 여러 정보를 함께 가지고 있다. 예를 들어, 함수 객체는 기본적인 전역 영역을 가지고 있고, 함수를 호출할 때 전달되지 않으면 자동으로 설정되는 인수의 기본값들을 가지고 있는 반면에 코드 객체는 그렇지 못하다. 또한, 코드 객체는 변경 불가능한 자료형의 일종이다.

속성 _code_를 이용하여 함수를 호출할 일은 거의 없지만, 유용한 정보를 추출해 내는 것은 가능하다. 다음은 함수의 이름과 인수의 개수, 지역 변수의 수, 지역 변수의 이름 등을 추출해 내는 예이다. 더 자세한 내용은 매뉴얼의 라이브러리 래퍼런스를 참고하기 바란다.

```
>>> def f(a, b, c, *args, **kw): #함수를 정의한다.
...     calx = 1
```

```
...     caly = 2
...     return 1
...
>>> code = f.__code__   # 코드 객체를 참조한다.
>>> code.co_name        # 함수의 이름
'f'
>>> code.co_argcount    # 필수적인 인수의 개수
3
>>> code.co_nlocals     # 전체 지역 변수의 수
7
>>> code.co_varnames    # 지역 변수의 이름들
('a', 'b', 'c', 'args', 'kw', 'calx', 'caly')
>>> code.co_code        # 코드 객체의 바이트 코드 명령어
b'd₩x01}₩x05d₩x02}₩x06d₩x01S₩x00'
>>> code.co_names
()
>>> code.co_filename    # 코드 객체를 포함하는 파일 이름
'<stdin>'               ---------- 대화 모드에서 수행하므로 stdin로 나온다.
>>> code.co_flags & 0x04        # 가변 인수를 사용하는가?
4
>>> code.co_flags & 0x08        # 키워드 인수를 사용하는가?
8
>>> code.co_flags & 0x20        # 발생자인가?
0
```

11.9 재귀적 프로그래밍

함수가 자기 자신을 호출하면 재귀적이라 한다. 재귀적 프로그래밍은 프로그래밍 언어에서 폭넓게 사용된다. 특히, 자연 언어 처리나 트리 탐색같은 분야에서는 자주 사용한다. 예를 들어, 1부터 N까지의 합을 계산하는 프로그램을 작성해 보자. 우선 합을 계산하는 재귀적 식(점화식)은 다음과 같이 주어진다.

```
sum(N) = N + sum(N-1), N > 1
sum(1) =1
```

1부터 N까지의 합은 N + (1부터 N-1까지의 합)과 같다. 초기 조건으로 sum(1)에 1이 할당되었다. 실제로 계산은 다음과 같이 수행한다.

```
sum(N) = N + sum(N-1)
= N + N-1 + sum(N-2)
= N + N-1 + N-2 + ... + 3 + 2 + 1
```

이것을 코드로 작성한 예이다.

```
>>> def sum(N):
...     if N == 1: return 1
...     return N + sum(N-1)
...
>>> sum(10)
55
```