단축 평가를 이용한 삼항연산자 <= 사용하지 말자

- and-or를 이용한 삼항 연산자, if-else를 이용한 삼항 연산자.

```
>>> a = 95
>>> print('학점:', (a>=90) and 'A' or 'B')          # (조건식) and 참 or 거짓
학점: A
>>> print('학점:', 'A' if a>=90 else 'B')           # 참 if (조건식) else 거짓
학점: A
```

- 문제의 경우

(a>=90)가 참이므로 and 결과는 뒤쪽 피연산자인 0
앞쪽 피연산자가 0이므로 or 결과는 뒤쪽 피연산자인 1.
최종적으로 (a>=90)이 참인데도 결과는 1.

```
>>> a = 95
>>> print('학점:', (a>=90) and 0 or 1)             # (조건식) and 참 or 거짓
학점: A
>>> print('학점:', 0 if a>=90 else 1)              # 참 if (조건식) else 거짓
학점: A
```

Chapter 3
함수

목차

- 함수의 정의
- 반환값
- 인자 전달
- 스코핑 룰
- 함수 인자
- lambda 함수
- 재귀적 함수 호출
- pass
- __doc__속성과 help함수
- 이터레이터(iterator)
- 제너레이터(generator)

함수의 정의

- 함수의 선언은 def로 시작하고 콜론(:)으로 끝냄
- 코드의 들여쓰기로 구분 → 시작과 끝을 명시해 줄 필요가 없음
- 헤더(header)파일을 따로 두거나
  인터페이스(interface)/구현(implementation)을 나누지 않음
- 함수 선언

```
def <함수명>(인수1, 인수2, ...인수N):
    <구문>
    return <반환값>
```

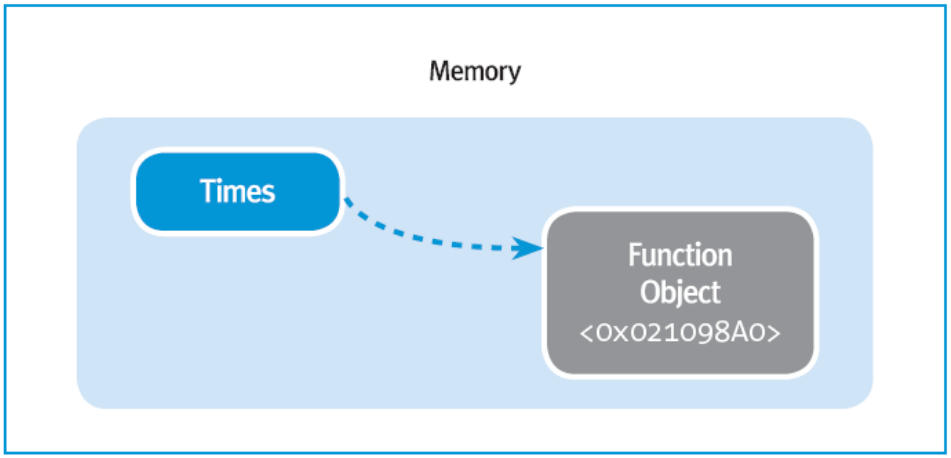
- 간단한 함수 : 2개 인수의 곱을 반환.

```
>>> def Times(a, b):
        return a*b

>>> Times          # Times는 함수 객체의 레퍼런스
<function Times at 0x000000000316E8C8>
>>> Times(10, 10)
100
```

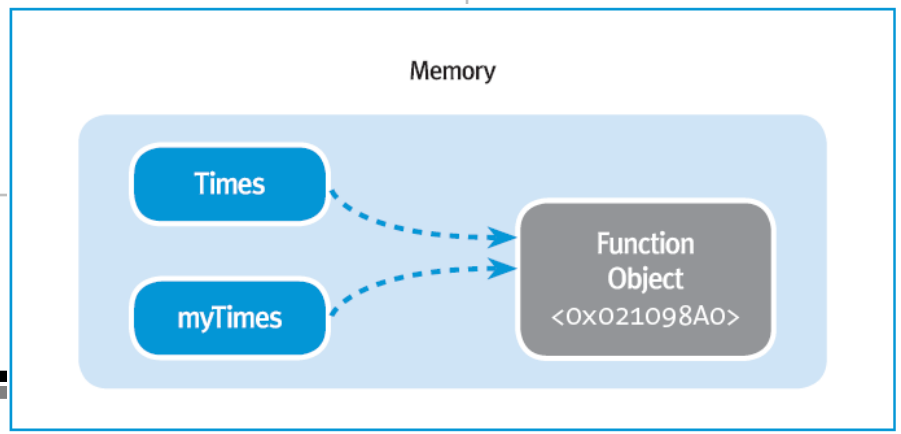
함수의 정의

- 함수정의 시 메모리에 생성되는 것.
  → (함수 객체) + (함수 객체를 가리키는 레퍼런스(함수이름))



- 함수 레퍼런스 복사 -> (함수 객체는 복사되지 않음)

```
>>> myTimes = Times #함수 레퍼런스 복사
>>> r = myTimes(10, 10)
>>> r
100
>>> globals()
```



내장 함수 목록 확인하기

- __builtins__ : 내장 영역의 이름들을 가진 module
- dir() 내장함수
  - 인자: 타입 또는 객체
    반환 : 네임스페이스에 등록된 모든 이름들(변수, 메소드)의 리스트.
  - 인자가 없으면 현재 영역이 포함한 변수, 메소드 리스트를 반환.

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__',]

>>> dir(__builtins__)  <= 모든 내장함수 보기
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError',
'DeprecationWarning', 'EOFError', 'Ellipsis', …… 'zip']
>>> "dir" in dir(__builtins__)  <= dir도 __builtins__에 소속되어 있음
True
```

return

- 함수를 종료하고 호출한 곳으로 돌아감.
- return 문이 없으면 None을 반환

```
>>> def setValue(newValue):
        x = newValue
>>> retval = setValue(10)
>>> print(retval)
None
```

- 하나의 값만 반환 가능. 여러 값을 반환하려면 하나의 튜플로 묶어서 반환하고, 받을 때는 여러 변수에 unpack하여 할당 가능.

```
>>> def swap(x, y):
        return y, x
>>> swap(1, 2)
(2, 1)
>>> a, b = swap(1, 2)
>>> a, b
(2, 1)
>>> x = swap(1, 2)
>>> type(x)
<class 'tuple'>
```

return

- 입력된 2개 리스트의 교집합 리스트 반환

```
>>> def intersect(prelist, postlist):
        retList = []                    # 교집합 리스트
        for x in prelist:
            if x in postlist and x not in retList:
                retList.append(x)
        return retList

>>> list1 = "SPAM"
>>> list2 = "EGG"
>>> intersect(list1, list2)
[]
>>> intersect(list1, ['H', 'A', 'M'])
['A', 'M']
>>> tup1 = ('B', 'E', 'E', 'F')
>>> intersect(list2, tup1)
['E']
```

인자 전달

- 함수는 네임스페이스를 생성함

```
>>> def fn():
...     num = 10
...     return num+2
...
>>> fn()
12
>>> print('num=',num)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'num' is not defined
```

함수 안에서 생성한 변수 num 은 별도의 네임스페이스에 있으므로 호출한 쪽에서는 볼 수 없음.

- (cf) 블록은 네임스페이스를 생성 안함

```
>>> if True:
...     mynum = 100
...
>>> print('mynum=',mynum)
mynum= 100
```

if 절 안에서 생성한 변수 mynum 은 동일한 네임스페이스에 있으므로 블록이 끝난 후에도 볼 수 없음.

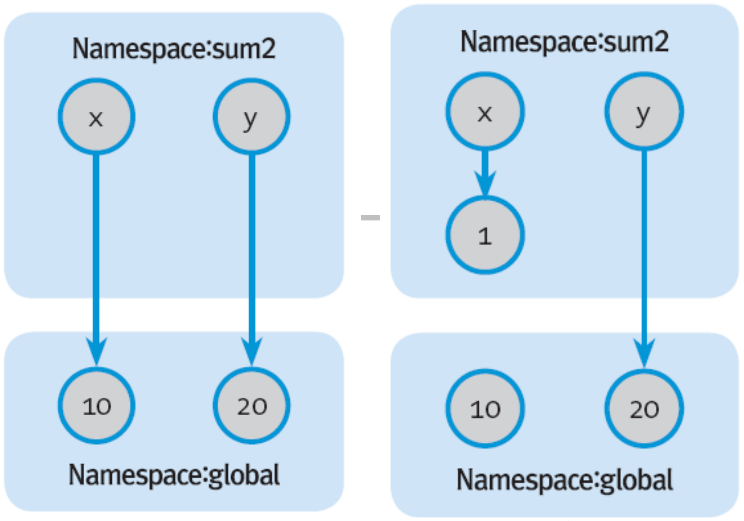
인자 전달

- 파이썬에서 **모든 함수 인자는 레퍼런스만 복사하여 전달**
  - 함수의 인자는 호출자 내부 객체의 레퍼런스

```
>>> def fn(n):
...     print('in fn():', id(n), n)
...
>>> a = 100
>>> id(a)
1500495392
>>> fn(a)
in fn(): 1500495392 100
```

인자 전달

- 함수 안에서 치환: 호출자가 전달하는 변수의 타입에 따라 다르게 처리
  - 변경가능 변수 (mutable)
  - 불가능 변수 (immutable)
- 함수 인자가 immutable일 때의 치환




```
>>> a = 10
>>> b = 20
>>> def sum1(x, y):
        return x + y

>>> sum1(a, b)
30
>>> x = 10
>>> def sum2(x, y):
        x = 1
        return x + y

>>> sum2(x, b)
21
>>> x
10
```

#값이 1인 새로운 객체 생성

#함수 내부의 변경사항이 외부에 영향 미치지 않음

인자 전달

- 함수 인자가 mutable일 때의 치환

```
>>> def change(x):
        x[0] = 'H'          # 객체 복사 없이 그대로 변경.

>>> wordlist = ['J', 'A', 'M']
>>> change(wordlist)        # change 함수가 wordllist를 변화시킴.
>>> wordlist
['H', 'A', 'M']
```

```
>>> def change(x):          # 호출자 쪽의 변수가 수정되지 않게 하려면
        x = x[:] # 입력받은 인자를 강제로 복사하여 사용.
        x[0] = 'H'
        return None

>>> wordlist = ['J', 'A', 'M']
>>> change(wordlist)        # change함수가 wordllist 를 변화시키지 않음.
>>> wordlist
['J', 'A', 'M']
```

스코핑 룰

- 이름공간 (Name Space) : 변수의 이름이 저장되어 있는 장소
  - 지역 영역(Local scope) : 함수 내부의 이름공간,
  - 전역 영역(Global scope) : 함수 밖의 영역,
  - 내장 영역 (Built-in Scope) : 파이썬 자체 정의 영역
- LGB 규칙 (이름을 검색하는 규칙)
  - Local Scope -> Global Scope -> Built-in Scope 순서로 찾음

```
>>> a = [1, 2, 3]              #전역영역 변수 a를 생성
>>> def scoping():             # 함수는 별도의 이름 공간을 가짐
        a = [4, 5, 6]          #지역영역 변수 a를 새로 생성

>>> a                          #전역변수 a는 안바뀜
[1, 2, 3]
```

스코핑 룰

- 지역영역에서 전역변수 "변경"하려면 global 사용
- global을 사용하지 않으면…
  - 함수 안에서 전역 변수를 read할 수는 있지만
  - 함수 안에서 전역 변수를 write할 수는 없다. 치환 연산을 하면 지역 영역에 새로운 객체와 레퍼런스가 생성되므로.

  => global로써 전역 변수 사용을 명시한 후에 write 가능.

```
>>> g = 1
>>> def testScope(a):
        global g            #전역변수 참조
        g = 2               #전역변수 변경
        return g + a

>>> testScope(1)
3
>>> g
2
```

스코핑 룰

- 내장영역 이름 리스트 : __builtins__

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
.....
'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super',
'tuple', 'type', 'vars', 'zip']

>>> x = [1, 2, 3]
>>> sum(x)                    # 내장 함수
6
```

min, max, len 등 유용한 내장함수들이 많다. 잘 봐두자.

함수 인자

- 기본 인자 값 : 함수 호출 시 인자를 주지 않았을 때 기본 값을 사용.
  - 함수 선언시, 기본 인자값은 맨 뒤부터 차례로만 줄 수 있음.

```
>>> def Times(a=10, b=20):        #기본인자 설정
        return a * b
>>> Times()
200
>>> Times(5)                      #a에 5할당
100
```

- 키워드 인자 (<=> 위치 인자): 인자 이름으로 특정 인자의 값 전달. 변수 전달 순서를 바꿀 수 있음
  - 함수 호출시, 키워드 인자는 맨 뒤부터 차례로만 사용할 수 있음.

```
>>> def connectURI(server, port):
        str = "http://" + server + ":" + port
        return str
>>> connectURI("test.com", "8080")
'http://test.com:8080'
>>> connectURI(port="8080", server="test.com") #순서상관없음
'http://test.com:8080'
```

함수 정의시 : 가변 개수의 위치인자 사용

- 가변인자 리스트: 인자개수가 미정, *를 인자 앞에 붙임
  - 호출할 때 **가변개수 인자** 사용 => 호출된 함수 안에서 **1개의 튜플**로 받아들임.

```
>>> def test(*args):                    #가변인자 리스트 args는 튜플로 처리
        print(type(args))

>>> test(1,2)
<class 'tuple'>


>>> def union2(*ar):      # 전달된 문자열들의 문자set을 리스트 타입으로 반환
    res = []
    for item in ar:          #튜플 ar에 들어있는 인자를 하나씩 얻어옴
      for x in item: #문자열에서 문자 하나씩 얻어옴
        if not x in res:
          res.append(x)
    return res

>>> union2("HAM", "EGG", "SPAM")
['H', 'A', 'M', 'E', 'G', 'S', 'P']
```

함수 호출시 : 리스트, 튜플, 셋, 딕셔너리의 원소 전달

- 호출할 때 군집자료형 앞에 * 를 붙이면 각각의 원소들이 인자로 전달됨.

```
>>> def test(*args):
...     print(type(args))
...     for i in args:
...         print('   ',i)
...
>>> l1 = [1,2,3]
>>> t1 = (1,2,3)
>>> s1 = {1,2,3}
>>> d1={"a":1,"b":2}

>>> fn(l1)
<class 'tuple'>
  [1, 2, 3]
```

1개의 리스트를 전달

```
>>> test(*l1)
<class 'tuple'>
   1
   2
   3
>>> test(*t1)
<class 'tuple'>
   1
   2
   3
>>> test(*s1)
<class 'tuple'>
   1
   2
   3
>>> test(*d1)
<class 'tuple'>
   a
   b
```

리스트의 각 원소를 전달

딕셔너리는 키값이 전달됨

함수 정의시 : 가변 개수의 키워드인자 사용

- 정의되지 않은 인자: **를 붙이면 가변길이 사전형식 인자 전달
  => 함수 안에서 **1개의 사전**으로 받아들임.
  - 호출 방법1 : **가변개수의 키워드 인자 형식**으로 호출
  - 호출 방법2 : 사전 객체 앞에 ** 붙여서 호출 (**호출할 때 딕셔너리 앞에 ** 를 붙이면 **각각의 키-값 쌍이 키워드 인자로서 전달**됨.)

keyword_parameter.py

```
def addItem( **values ):
    print('values:', end=' ')
    for k, v in values.items():
        print(k, v, ',', end=' ')
    print("")

addItem( name="홍길동", tel="010-2222-3333" )

item = {"name": "홍길동", "tel": "010-2222-3333"}
addItem( **item )
```

[실행결과]
values: name 홍길동 , tel 010-2222-3333 ,
values: name 홍길동 , tel 010-2222-3333 ,

함수 정의시 : 가변 개수의 키워드인자 사용 예

- url 인자 전달에 활용예.
  - 아래와 같은 [실행결과]가 나오도록 userURIBuilder 함수를 작성해 보자.

[실행결과]
http://test.com:8080/?id=userid&passwd=1234&
http://test.com:8080/?id=userid&passwd=1234&name=mike&age=20&

람다(lambda) 함수

- 이름없는 함수객체, 1줄 짜리 함수, return 문 없이 유일한 수행문의 결과가 자동으로 반환됨.

```
lambda 인수 : <구문>
```

\(backslash) 이용하여 여러 줄로 작성 가능하지만 가독성을 위해 한 줄로만 사용하자.

- 용도
  - 한 줄의 간단한 함수가 필요한 경우, 프로그램의 가독성을 위해서
  - 함수를 인자로 넘겨 줄 때

```
>>> g = lambda x, y : x * y        # 람다함수를 g에 대입
>>> g(2, 3)
6
>>> (lambda x: x * x)(3) # 이 함수 객체는 사용 후 바로 삭제됨
9
>>> globals()                      # 전역 영역에 g만 존재
{'__doc__': None, 'g': <function <lambda> at 0x000000000313E8C8>,
'__builtins__': <module 'builtins' (built-in)>, '__spec__': None, '__package__':
None, '__name__': '__main__', '__loader__': <class
'_frozen_importlib.BuiltinImporter'>}
```

람다(lambda) 함수와 내장함수 map

- 내장 함수 map에 사용되는 예
  - map 함수 : 순회 가능한 객체의 각 원소를 입력 받은 함수에 하나씩 적용한 결과를 반환.

```
>>> def sqrt(x):
...     return x**2
...
>>> list(map(sqrt, [1,2,3]))
[1, 4, 9]
>>>
>>> list(map(lambda x: x**2, [1,2,3]))
[1, 4, 9]
>>> it = map(lambda x: x**2, [1,2])
>>> next(it)
1
>>> next(it)
4
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

sqrt 함수를 만들고 이를 map함수에 전달.

lambda함수를 map함수에 전달.

map, filter 함수의 결과는 iterator 객체.
-> next()로써 차례로 원소를 꺼낼 수 있다.

람다(lambda) 함수와 내장함수 filter

- 내장 함수 filter에 사용되는 예
  - filter 함수 : 순회 가능한 객체의 각 원소에 함수를 적용한 결과가 참인 원소들로 이루어진 iterator 객체를 반환.

```
>>> l1 = [1,2,3]
>>> filter(lambda i: i <= 2, l1)
<filter object at 0x00DC0568>
>>> list(filter(lambda i: i<= 2, l1))
[1, 2]
>>> tuple(filter(lambda i: i<= 2, l1))
(1, 2)
```

반환값이 bool인 함수 사용.

Iterator 객체이므로 list나 tuple로 전환하여 사용 가능.

재귀 함수

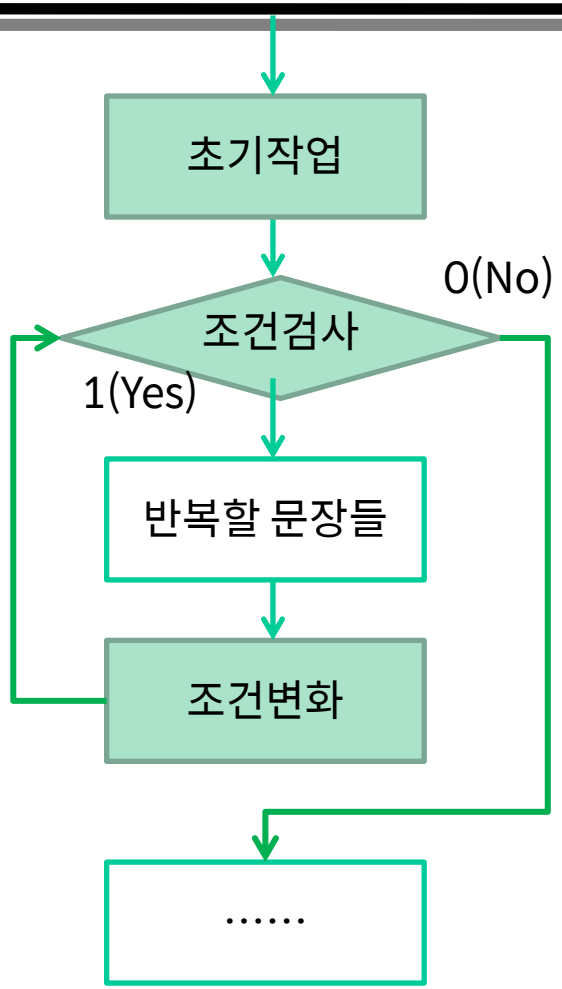
[재귀함수란] : 자신을 호출하는 함수

[재귀함수 작성 방법]

(1) 초기작업 :
외부에서 재귀함수 호출시의 인자값

(2) 조건변화 :
재귀 호출시의 인자값 변화

(3) 조건검사 :
최소한의 문제일 때 재귀호출 없이 반환하기



재귀적 함수 호출 – factorial

- X! 값 구하기

```
>>> def factorial(x):
        if x == 1:
            return 1
        return x * factorial(x - 1)

>>> factorial(10)
3628800
```

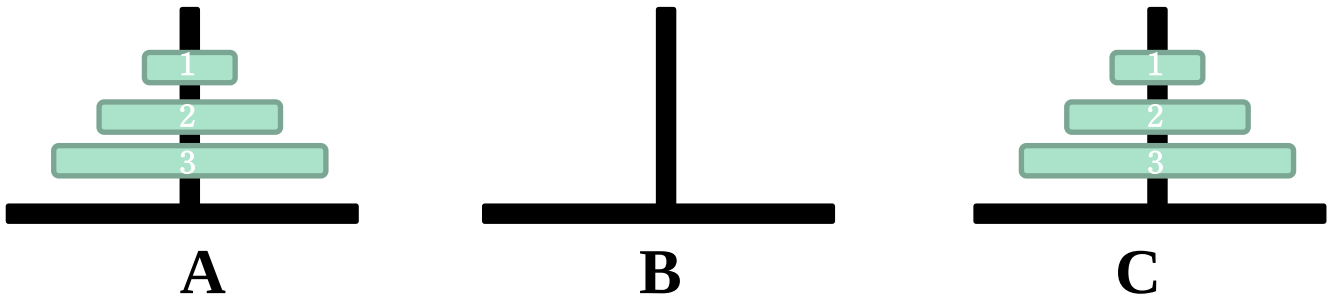
조건검사

조건변화

초기작업

하노이 탑 문제 (https://ko.wikipedia.org/wiki/ 하노이의_탑)

- 세 개의 기둥과 이 기둥에 꽂을 수 있는 서로 크기가 다른 원판들이 있고, 시작 시점에는 한 기둥에 모든 원판들이 큰 것부터 작은 것 순으로 쌓여 있다. 한 기둥에 꽂힌 원판들을 그 순서 그대로 다른 기둥으로 옮겨서 다시 쌓아 보자.
- 조건
  - 원반은 한 번에 하나씩만 옮길 수 있다.
  - 옮기는 과정에서 작은 원반의 위에 큰 원반이 올려져서는 안 된다.



재귀적 함수 호출 - hanoi tower

hanoi.py

- 하노이의 탑
- 아래와 같은 [실행결과]가 나오도록 hanoi 함수를 작성해 보자.

[실행결과]

```
1 : A -> C
2 : A -> B
1 : C -> B
3 : A -> C
1 : B -> A
2 : B -> C
1 : A -> C
```

pass 구문

- 아무 것도 하지 않는 구문
- pass가 필요한 이유
  - : (colon) 으로 시작되는 블록은 비워놓을 수 없으므로 프로토타입 단계에서 함수, 모듈, 클래스의 이름만 만들어 놓으려 할 때 사용

```
>>> class temp:      #일단 클래스 생성하고 나중에 변수, 메소드 추가
        pass
```

  - 아래 예와 같이 아무 것도 하지 않는 블록이 필요할 때 pass를 이용하면 의미 없는 작업을 시키지 않아도 됨.

```
>>> while True:          #Ctrl+C 누를 때까지 종료되지 않음
        pass
                         #Ctrl+C 누름
Traceback (most recent call last):
  File "<pyshell#28>", line 2, in <module>
    pass
KeyboardInterrupt
```

__doc__속성과 help 함수

- help함수 : __doc__ 속성 (모든 객체의 부모인 object의 기본속성) 활용
  - 내장함수들은 내용을 이미 갖고 있음
  - 사용자 정의 함수의 __doc__ 내용 : 기본으로 함수 헤더를 보여줌

```
>>> help(print)  # 내장함수 print의 document 요청
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
>>>
>>> def plus(a, b): # 사용자 함수
        return a + b
>>> help(plus)      # 사용자 함수의 기본 help
Help on function plus in module __main__:

plus(a, b)
```

기본 help()로써 함수 헤더만 보임

__doc__속성과 help 함수

- help함수 : __doc__ 속성 (모든 객체의 부모인 object의 기본속성) 활용
  - 사용자 정의 함수에서 __doc__ 속성 이용 방법

```
>>> plus.__doc__="return the sum of parameter a, b " # (1) __doc__값 변경
>>> help(plus)
Help on function plus in module __main__:

plus(a, b)
    return the sum of parameter a, b

>>> def plus(a, b):      # (2) 함수 시작 부분에 문자열을 두어 __doc__값 주기.
...     """두 수의 합을 반환
...     >>> plus(10,20)
...     """
...     return a+b
...
>>> help(plus)
Help on function plus in module __main__:

plus(a, b)
    두 수의 합을 반환
    >>> plus(10,20)
```

이터레이터(iterator)

- 순회가능한 객체(리스트, 튜플, 문자열 등) 요소에 순서대로 접근할 수 있는 객체
- for문이 순회가능 객체의 모든 요소에 접근하는 방법 : iterable 객체의 **__iter__()**, **__next__()** 메소드를 이용 → 한 번 호출할 때마다 첫 요소부터 StopIteration 예외를 만날 때까지 모든 원소를 순회.

```
>>> for element in [1, 2, 3]:          #리스트 순회
        print(element)
1
2
3
>>> for element in (1, 2, 3):          #튜플 순회
        print(element)

>>> for key in {'one':1, 'two':2}:     #사전 키값 순회
        print(key)
two
one
>>> for ch in "123":                   #문자열 한문자씩 순회
        print(ch)
>>> for line in open("myfile.txt"):    #파일 내용 순회
        print(line)
```

이터레이터(iterator)

- 첫 요소부터 차례로 한 요소씩 가져오는 2가지 방법
  - iterable 객체의 메소드 **__iter__(), __next__()**
  - 내장함수 **iter()**, **next()**
- 내장함수 **iter() :** 순회가능한 객체에서 이터레이터 객체(1회용) 가져옴
- 내장함수 **next() :** 이터레이터 객체의 __next__()를 호출

```
>>> s1 = 'ab'
>>> it = s1.__iter__()
>>> it.__next__()
'a'
>>> it.__next__()
'b'
>>> it.__next__()
Traceback (most recent call
last):
  File "<stdin>", line 1, in
<module>
StopIteration
```

```
>>> s1 = 'ab'
>>> it = iter(s1)
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
Traceback (most recent call
last):
  File "<stdin>", line 1, in
<module>
StopIteration
```

제너레이터 (generator)

- return 대신 yield를 가진 함수: yield에서 값이 반환되지만 함수가 스택에 실행되던 상태 그대로 보존되었다가 next()를 호출하면 중단 지점부터 이어서 수행됨.
  - **range(시작, 종료, 증감)** : 수열 생성, 시작값 포함 O, 종료값 포함 X
  - 함수의 상태가 보존되어 **for**문의 순회 **index**가 초기화되지 않아서 순서대로 반환됨 : 순회가능 객체, 제터레이터객체

```
>>> def reverse(data):
        for index in range(len(data) - 1, -1, -1):
            yield data[index]                    #3, 2, 1, 0

>>> for ch in reverse("golf"):
        print(ch)

f
l
o
g
```

[generator의 장점]
필요한 값을 메모리에 저장해 두고 사용하는 것이 아니라 계산에 의해 연속으로 만들어 사용하므로 메모리 절약 효과. => 다음 장의 예제 참조

제너레이터 (generator)

fibo_generator.py

- 피보나치 수열 (0, 1, 1, 2, 3, 5, 8, …)
  - F(0)=0, F(1)=1, F(n) = F(n-2)+F(n-1) if n>1
- **enumerate() 내장함수**: 순회 가능 객체에서 인덱스와 요소를 둘 다 반환

```
def Fibonacci():
    a, b = 0, 1
    while 1:
        yield a      # a,b 값이 초기화되지 않고 갱신되어 반환됨
        a, b = b, a + b

for i, ret in enumerate(Fibonacci()): #인덱스와 요소값
    if i < 20: print(i, ret)
    else: break
```

[실행결과]

```
0 0
1 1
2 1
3 2
4 3
5 5
6 8
7 13
8 21
9 34
```

List Comprehension (리스트 내장)

- 리스트의 각 원소에 일정한 식을 적용한 리스트를 생성한다.
- 문법: [(변수를 활용한 식) for (변수 이름) in (순회할 수 있는 값)]

숫자의 빈도수 구하기

- 1과 100 사이의 정수를 입력 받고 각 정수의 빈도수를 세는 프로그램을 작성하시오.

```
1과 100 사이의 정수를 입력하세요: 2 5 6 5 4 3 23 43 2 Enter
2 - 2번 나타납니다.
3 - 1번 나타납니다.
4 - 1번 나타납니다.
5 - 2번 나타납니다.
6 - 1번 나타납니다.
23 - 1번 나타납니다.
43 - 1번 나타납니다.
```

제어문과 연관된 유용한 함수

range() : 수열의 생성에 사용

- range([start], stop[,step])
  - stop값은 필수, start(기본 0)과 step(기본 1)은 옵션
  - start ~ (stop-1) 까지 차례로 생성. stop값은 포함되지 않음.

```
>>> list(range(10))          #stop만 있음. 10은 포함되지 않음
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(range(5,10))        #start, stop
[5, 6, 7, 8, 9]

>>> list(range(10,0,-1))     #start, stop, step
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

>>> list(range(10,20,2))
[10, 12, 14, 16, 18]
```

range() : 수열의 생성

- **range()** 함수를 이용해 **10**에서 **20**까지 짝수 출력 예제

리스트 항목과 인덱스 값을 동시에 얻는 방법

- len()함수 이용 ← C언어 스타일로 사용한 예.

```
>>> L = ['Apple', 'Orange', 'Banana']
>>> for i in range(len(L)):              #len() 시퀀스 원소 개수 출력
        print("Index: {0}, Value: {1}".format(i, L[i]))


Index: 0, Value: Apple
Index: 1, Value: Orange
Index: 2, Value: Banana
```

리스트 항목과 인덱스 값을 동시에 얻는 방법

enumerate(sequence object[, start=0]) 내장 함수 이용
(start(기본 0)는 index의 시작값으로서 생략가능)

- 튜플 (인덱스 값, 항목 값)을 반환하므로 unpack하여 사용하면 편리.

```
>>> L = [100, 15.5, "Apple"]
>>> for i, data in enumerate(L):
        print(i, data)
0 100
1 15.5
2 Apple

>>> for i, v in enumerate(L,101):  #start 지정
        print(i, v)
101 100
102 15.5
103 Apple
```

리스트 내장(list comprehension)

[<expression> for <item> in <sequence object> (if <condition>)]

- 기존 리스트 객체를 이용해 조합, 필터링 등의 연산을 통해 새로운 리스트 객체 생성

```
>>> l = [1,2,3,4,5]                          #리스트 객체
>>> [i ** 2 for i in l]
[1, 4, 9, 16, 25]

>>> t = ("apple", "banana", "orange")        #튜플 객체
>>> [len(i) for i in t]                      #문자열의 길이 리스트
[5, 6, 6]

>>> d = {100:"apple", 200:"banana", 300:"orange"} #사전
>>> [v.upper() for v in d.values()]          #사전 문자열을 대문자 리스트
['BANANA', 'ORANGE', 'APPLE']

#range() 이용하여 원하는 수열값을 가지는 리스트 생성
>>> [i**3 for i in range(5)]
[0, 1, 8, 27, 64]
```

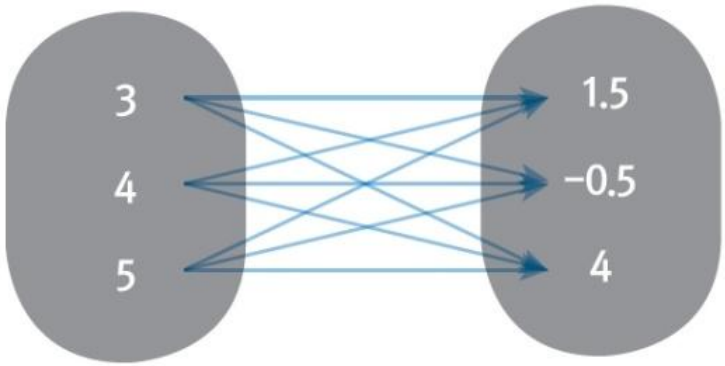
리스트 내장(list comprehension)

- if <condition> 사용

```
>>> l = ["apple", "banana", "orage", "kiwi"]
>>> [i for i in l if len(i) > 5]          #문자열 길이가 5 이상인 리스트
['banana', 'orage']
```

- 2개 이상 리스트의 조합

```
>>> L_1 = [3,4,5]
>>> L_2 = [1.5, -0.5, 4]
>>> [x * y for x in L_1 for y in L_2]
[4.5, -1.5, 12, 6.0, -2.0, 16, 7.5, -2.5, 20]
```



- input() 함수와 연결 -> 여러 숫자를 입력 받아 숫자 리스트로 만들기.

```
>>> L = [eval(s) for s in input('정수들 입력:').split()]
정수들 입력:1 2 3 99 100
>>> L
[1, 2, 3, 99, 100]
```

filter() 함수

filter (<function>|None, iterable)

- 함수의 결과가 참인 리스트 객체의 이터레이터 반환
- 함수 대신 None이 오는 경우 필터링 하지 않음

```
>>> L = [10, 25, 30]
>>> Iter = filter(None, L)                    # None이므로 필터링 안함
>>> for i in Iter:  # 이터레이터 객체이므로 for문으로써 값을 확인 가능
        print("Item: {0}".format(i))
Item: 10
Item: 25
Item: 30
>>> def GetBiggerThan20(i):                   # True/False 반환 필터링함수
        return i > 20
>>> IterL = filter(GetBiggerThan20, L)        #필터링함수 사용
>>> for i in IterL:
        print("Item: {0}".format(i))
Item: 25
Item: 30
```

filter() 함수

filter (<function>|None, iterable)

- 기존 객체 L은 바뀌지 않음
- 이터레이터 객체를 list() 생성자로써 리스트로 바꿈.

```
>>> NewL = list(filter(GetBiggerThan20,L))
>>> NewL
[25, 30]
>>> L
[10, 25, 30]
```

- lambda 함수 사용 (return 없어도 반환됨)
- 이터레이터 객체이므로 for문으로써 값을 확인.

```
>>> IterL = filter(lambda i: i>20, L)
>>> for i in IterL:
        print("Item: {0}".format(i))

Item: 25
Item: 30
```

zip() 함수

- zip()
  - 2개 이상 시퀀스형이나 이터레이터형 객체를 튜플 형태로 묶음
  - 반환값은 튜플 객체의 이터레이터

```
>>> X = [10, 20, 30]
>>> Y = ['A', 'B', 'C']
>>> for i in zip(X,Y):
        print("Item: {0}".format(i))
Item: (10, 'A')
Item: (20, 'B')
Item: (30, 'C')
```

- zip함수 결과값이 이터레이터이므로 결과를 객체에 저장하려면 list(), tuple() dict() 등 생성자 이용

```
>>> RetList = list(zip(X, Y))
>>> RetList
[(10, 'A'), (20, 'B'), (30, 'C')]
```

zip() 함수

- zip() 으로써 군집 자료형의 각 원소로부터 같은 위치의 값 끼리 분리할 수 있음

```
>>> X2, Y2 = zip(*RetList)
>>> X2
(10, 20, 30)
>>> Y2
('A', 'B', 'C')
>>> x, y = zip(*[(10, 'A'), (20, 'B'), (30, 'C')])
>>> x
(10, 20, 30)
>>> y
('A', 'B', 'C')
```

- 3개 이상의 객체도 결합 가능. 인자 개수가 동일하지 않으면 짧은 쪽에 맞춤

```
>>> X = [10, 20, 30, 40]
>>> Y = "ABC"
>>> Z = (1.5, 2.5, 3.5, 4.5)
>>> RetList = list(zip(X,Y,Z))
>>> RetList
[(10, 'A', 1.5), (20, 'B', 2.5), (30, 'C', 3.5)]        # 원소가 3개까지만 생성됨.
```

map() 함수

map (<function>, iterable)

- 순회가능 객체의 각 아이템을 함수에 전달하고 그 결과를 이터레이터 객체로 반환

```
>>> L = [1, 2, 3]
>>> def Add10(i):
        return i + 10
>>> for i in map(Add10, L):        #값을 갱신하므로 filter()와 다르다
        print("Item: {0}".format(i))
Item: 11
Item: 12
Item: 13
```

- (함수가 간단하다면) Lambda함수 사용
- 이터레이터 객체를 list()생성자로써 리스트로 바꿈.

```
>>> RetList = list(map((lambda i: i+10), L))
>>> RetList
[11, 12, 13]
```

map() 함수

map (<function>, iterable)

- 2개 이상 인자를 가지는 함수는 시퀀스 객체도 2개 이상

```
>>> X = [1,2,3]
>>> Y = [2,3,4]
>>> RetList = list(map(pow, X, Y)) #pow 함수는 인자가 2개
>>> RetList
[1, 8, 81]
```

- map()을 input() 함수와 연결

  eval: 문자열 형태의 식을 실행시켜 그 결과를 반환.

```
>>> L = list(map(eval, input('정수들 입력:').split()))
정수들 입력:1 2 3 99 100
>>> L
[1, 2, 3, 99, 100]
```

효율적인 순회 방법

- 시퀀스형 자료를 순회하며 출력하는 방법
  - for 문 이용.

```
>>> l = ['Apple', 'Orange', 'Banana']
>>> for i in l:
        print(i)              # print 함수를 아이템 개수만큼 호출
Apple
Orange
Banana
```

  - join() 함수 (시퀀스형 객체 아이템을 주어진 문자열로 연결) 이용.

```
>>> print("\n".join(l))     # print 함수를 1번 호출 (더 빠름)
Apple
Orange
Banana
```

연습문제

- 임의 개수의 숫자를 읽어 들인 후 입력한 수의 총 개수와 10보다 큰 수의 개수를 출력하시오.
- [힌트] split, eval, 리스트 내장 활용.

[수행결과]
숫자 입력: 25 9 99 80 1 2 3 4 5
입력한 갯수는 9개이고 10보다 큰 수는 3개이다